

### 版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



O'REILLY®

仅供非商业用途或交流学习使用

第2版

构建健壮的数据中心



# 高可用MySQL

MySQL High Availability, Second Edition

[美] Charles Bell, Mats Kindahl, Lars Thalmann 著  
Pinterest technologists 作序  
宁青 唐李洋 译



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>



O'REILLY®

# 高可用MySQL

## (第2版)

MySQL High Availability, Second Edition

Charles Bell

[美] Mats Kindahl 著

Lars Thalmann

Pinterest technologists 作序

宁青 唐李洋 译

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING





## 内 容 简 介

本书主要讲解真实环境下如何使用MySQL的复制、集群和监控特性，揭示MySQL可靠性和高可用性的方方面面。本书定位于解决MySQL数据库的常见应用瓶颈，在保持MySQL持续可用性的前提下，挖潜各种提高性能的解决方案。本书描述了很多MySQL工具的变化，涵盖了5.5版本的知识，以及若干5.6版本的功能。本书的作者正是书中介绍的很多工具的设计师，本书揭示了MySQL可靠性和高可用性的许多不为人知的方面。

本书适用于MySQL数据库管理员及MySQL应用开发者。对于相关专业的师生，本书也有很高的参考价值。

©2014 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2015. Authorized translation of the English edition, 2014 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有出版权由O'Reilly Media, Inc.授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2015-2367

## 图书在版编目 ( CIP ) 数据

高可用MySQL：第2版 / (美) 贝尔 (Bell, C.)，(美) 肯德尔 (Kindahl, M.)，(美) 塞尔蒙 (Thalmann, L.) 著；宁青，唐李洋译. —北京：电子工业出版社，2015.10

书名原文：MySQL High Availability, Second Edition

ISBN 978-7-121-26688-1

I. ①高… II. ①贝… ②肯… ③塞… ④宁… ⑤唐… III. ①关系数据库系统 IV. ①TP311.138

中国版本图书馆CIP数据核字 (2015) 第164441号

策划编辑：张春雨

责任编辑：刘 舫

封面设计：Karen Montgomery 张 健

印 刷：三河市君旺印务有限公司

装 订：三河市良远印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱

邮编：100036

开 本：787×980 1/16

印张：43.75

字数：980千字

版 次：2015年10月第1版

印 次：2017年12月第4次印刷

定 价：128.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。



# 译者序

MySQL 是最受欢迎的开源数据库，她拥有相当大的装机量。而且 DB-Engines 的排名一直处于数据库总榜第二名的位置，仅次于 Oracle。MySQL 在开源领域排名第一，而第二大开源数据库 PostgreSQL 的分数仅仅是 MySQL 的零头。

MySQL 拥有庞大的用户群，国外的有 Facebook、Flickr、eBay 等，国内的有阿里、腾讯、新浪、百度等。而这些互联网和大部分传统公司的服务需要 7 × 24 小时连续工作。当此类型网站的部分数据库服务器宕机时，就需要高可用技术将流量牵引至备份主机，从而对在线业务产生尽可能少的影响甚至没有影响。

此时这些公司需要通过备份和恢复手段来产生备机，并通过复制来同步主备机间的状态，同时部署各种监控软件来监控服务器状态。当异常数据库服务器宕机时，通过手工或自动化手段将主机流量切换至备机，这个动作叫作 failover。而一些大型公司在面对成千上万台 MySQL 服务器时，通常使用自动化运维脚本或程序完成上述种种动作。

本书解决的是 MySQL 高可用问题，并围绕着高可用问题从复制、备份恢复、监控和自动化运维 4 个方面的知识点入手。无论你的应用是迷你型的博客型应用，还是 BAT 这种超大型互联网应用，本书所涵盖的知识点均适用。

接触上一版的时候还是 2010 年，转眼 5 年过去了，MySQL 也从 5.1 升级到 5.6，运维工具和运维方式都有较大的变化。第二版也与时俱进地增加了一些实用性章节，本书是了解和学习 MySQL 高可用技术相对来说较为经典的一本好书。在翻译过程中，我们努力体现原作者想表达的意思，但由于水平有限，有些遣词造句还是无法达到“信达雅”，且疏漏在所难免，恳请读者批评指正。我的微博：<http://weibo.com/ninqing>，可随时与我联系。





这本书还是由唐李洋和我共同翻译，翻译过程由于工作原因拖延不少时间，感谢张春雨和刘舫几位老师的辛苦工作和耐心等待。还要感谢我在平安的同事，汪洋、王鹏冲、张建龙、黄建蝉、王强、张阳，啥都不说了。最后感谢我的爱人王新，女儿宁悦晗，还有3个月后见面的家庭新成员。

宁青

2015年8月27日 于深圳观澜

宁青，资深 MySQL 数据库专家，现任平安科技高级数据库架构师，资深研究员。目前作为平安科技数据库部门 MySQL 负责人，负责制订平安科技的 MySQL 规范、流程和标准等。

唐李洋，合肥工业大学管理科学与工程博士。研究领域为大数据、数据挖掘与商务智能，现就职于中国电子科技集团公司第三十八研究所，公共安全技术研究院工程师。



## 第 2 版序

2011 年，Pinterest 开始发展起来。有人说我们比目前其他任何创业公司的发展都要快。刚开始，我们每天都要面临一个新的扩展性瓶颈，它会拖慢整个网站甚至搞垮一切。还记得我们无论去哪里都要带上笔记本电脑，那时我们的脑子里深深印着那些停机警告的短信声音。

当基础设施不断地被逼到极限的时候，你就不得不寻求另一种简单的出路。在成长的过程中，我们尝试了至少 5 种广为人知的数据库技术，它们都声称能够解决我们所有的问题，可每一次都灾难性地失败了，除了 MySQL。那是 2011 年 9 月，我们决定从头再来。我们用 MySQL、Memcache 和 Redis 对一切进行了重新设计，只有三个工程师而已。

MySQL？为什么是 MySQL？对每一种技术，我们都考虑了其最大关注点，并提出同样的问题。下面是我们对 MySQL 的考虑：

- 它解决了我们的存储需求吗？没错，我们需要映射、索引、排序和 blob 存储，这些 MySQL 都有。
- 它常用吗？你可以招聘到相关员工吗？MySQL 是目前生产线上最常使用的数据库之一。很容易招到使用过 MySQL 的人，我们可以到帕罗奥多市外走走，大喊我们需要 MySQL 工程师，就会冒出来好几个。这可不是开玩笑的。
- 它的社区活跃吗？非常活跃。有好多非常棒的书籍，和一个强大的在线社区。
- 面对故障，它健壮吗？即使在最恶劣的情况下，我们也从来没有丢失过数据。
- 它的扩展性如何？就它本身来说，只是一个很小的组件。我们需要一种上层的分片方案（这完全是另一个问题）。
- 你会是最大的用户吗？不，目前不是。最大的用户包括 Facebook、Twitter 和 Google。除非你能够改进一种技术，否则你不会想要成为它最大的用户。如果你是最大的用户，你会碰到一些新的扩展性问题，而其他人根本没机会遇到。
- 它的成熟度如何？真正的区别在于成熟度。根据复杂度的不同，成熟度就好比衡量完成一个程序所需的血、汗和泪。MySQL 的确复杂，但比不上那些神奇的自动集群 NoSQL 方案。而且，MySQL 拥有 28 年最好和最聪明的贡献，来自于诸如 Facebook 和 Google 那样大规模使用它的公司。根据我们的成熟度定义，在我们审





查的所有技术中，MySQL 是一个明智的选择。

- 有好的调试工具吗？作为一个成熟的产品，你当然需要强大的调试和分析工具，因为人们很容易遇到一些类似的棘手情况。比如你可能在凌晨三点遇到问题（不止一次）。相比用另一种技术重写一遍熬到凌晨六点，发现问题的根源然后回去睡觉舒服多了。

我们调查了差不多 10 种数据库技术后发现选择 MySQL 是一个明智的选择。MySQL 很棒，但它好比不给你任何行李就把你丢到目的地，让你不得不自食其力。它运行顺利的时候你可以连接到它，但一旦你开始使用它进行扩展，问题便开始满天飞：

- 我的查询执行很慢，怎么办？
- 我是不是应该启用压缩？怎么做呢？
- 扩展有哪些方法？
- 怎样复制？主 - 主复制（master-master replication）怎样？
- 复制停止了！怎么办？
- 持久性（durability，即 fsync 速度）有哪些选项？
- 我的缓冲区应该设为多大？
- *mysql.ini* 文件里有那么多选项，它们是什么意思？应该怎么设置？
- 我刚刚不小心写到 slave 里面去了！怎么防止下次发生同样的事情？
- 如何防止不带 where 子句的 update 命令执行？
- 应该用什么调试和分析工具？
- 要使用 InnoDB、MyISAM 或者其他存储引擎吗？

虽然可以通过在线社区查到问题答案、找到范例、修复漏洞，以及提供解决方法，但通常缺乏强大的凝聚力，而关于架构的深层讨论更是寥寥无几。我们已经知道如何小规模地使用 MySQL，但这种规模和步调简直是在开玩笑。本书可帮助我们更深刻地了解 MySQL。

MySQL 5.6 有一个新特性，即全局事务处理（Global Transaction Handlers），为复制树（replication tree）中的每个事务添加一个唯一标识。这个新特性使故障转移和 slave 提升变得容易很多。为此我们等了太久，终于在新版本中很好地实现了。

当我们采用分片方案进行重大的重构时，关于架构决策问题我们参考了本书，比如复制技术和拓扑、数据分享方案、监测、调整以及云相关的问题等。它让我们更深刻地理解了 MySQL 的底层运作，使我们更加了解了高级查询、访问模式、使用什么结构，以及之后的重复设计。时至今日，MySQL 架构仍然为 Pinterest 的核心数据服务。

——Yashwanth Nelapati 和 Marty Weiner

Pinterest

2014 年 2 月



# 第 1 版序

关于复制（Replication）的研究很多，但其中的大多数研究成果都没有得到应用。相反，MySQL 复制已经被广泛部署，但其原理并不为大多数人所知，本书将改变这种状况。本书中介绍的内容比较适合以下人群：愿意阅读大量的源代码，而且在生产环境中花很多时间进行调试，能够在深夜会议中探讨这些内容的人。

复制允许在出现不可避免的故障的情况下提供高可用的数据服务。故障的原因很多，包括磁盘、服务器或数据中心的故障。即使所有硬件都是完美无缺且完全冗余的，还有人为主观因素的影响。例如，数据库表可能被误删，应用程序可能写入了不正确的数据等，总会有偶然故障发生。但通过合理的准备工作，可以保证从故障中恢复，关键是冗余和备份。MySQL 复制支持冗余和备份。

但 MySQL 的复制并不仅限于支持故障恢复，它还频繁用于读操作的横向扩展（scale out）。MySQL 可以实现大量服务器的高效复制。对于那些读频繁的应用，在商用硬件上支持大量查询是一个低成本且有效的策略。

MySQL 复制还有其他有用的应用。在线数据定义语言（DDL）是关系型数据库管理系统中非常复杂的一个特性。MySQL 不支持在线 DDL（5.6 版本已经支持），但通过使用复制，往往可以足够好地部分实现它。如果有创意，还可以使用复制做更多的事情。

复制是使得 MySQL 如此广泛流行的特性之一，它允许将流行的 MySQL 原型转换为成功的商业关键部署。复制主张简单和便于使用，这一点和 MySQL 十分相似。然而，在生产环境中运行得往往不够完美。本书解释了成功使用 MySQL 复制所必须知道的内容，帮助读者理解复制是怎样实现的，哪些地方可能出错，怎样防止问题的出现，以及怎样在问题出现的时候解决它们——尽管你已经很努力地避免这些问题。

MySQL 复制还在继续完善中。与故障一样，变化总是存在的。MySQL 需要不断应对这些变化，使得复制更高效、更健壮、更有趣。例如，基于行的复制（row-based replication）是 MySQL 5.1 中的新特性。





尽管 MySQL 部署形态各异，规模各不相同，我最关心的还是互联网应用的数据服务。MySQL 到分布式存储系统（如 HBase 和 Hadoop）复制的可能性也使我兴奋不已。这样 MySQL 就可以更好地共享数据中心。

我曾经在 Facebook 和 Google 的团队支持重要的 MySQL 部署，有机会和时间学习这本书中所覆盖的很多东西。本书的作者们同样是 MySQL 复制的专家，通过阅读这本书，读者可以分享他们的专业知识。

——Mark Callaghan



# 目录

前言 .....	xxi
----------	-----

## 第 1 部分 高可用性和可扩展性

第 1 章 引言 .....	2
到底什么是复制 .....	4
那么，是否需要备份 .....	5
什么是监控 .....	6
其他阅读材料 .....	6
小结 .....	7
第 2 章 MySQL Replicant 库 .....	8
基本类和函数 .....	12
对各种操作系统的支持 .....	13
服务器 .....	13
服务器角色 .....	15
小结 .....	17
第 3 章 MySQL 复制原理 .....	18
复制的基本步骤 .....	19
配置 master .....	20
配置 slave .....	21
连接 master 和 slave .....	22
二进制日志简介 .....	23
二进制日志记录了什么 .....	24



观察复制的动作 .....	25
二进制日志的结构和内容 .....	27
建立新 slave .....	30
克隆 master .....	31
克隆 slave .....	33
克隆操作的脚本 .....	35
执行常见的复制任务 .....	37
报表 .....	37
小结 .....	43
<b>第 4 章  二进制日志 .....</b>	<b>45</b>
二进制日志的结构 .....	46
binlog 事件的结构 .....	48
事件校验 .....	50
将语句写入日志 .....	51
写入 DML 语句 .....	52
写入 DDL 语句 .....	52
写入查询 .....	52
LOAD DATA INFILE 语句 .....	58
二进制日志过滤器 .....	60
触发器、事件和存储例程 .....	62
存储过程 .....	68
存储函数 .....	70
事件 .....	74
特殊结构 .....	75
非事务型变更和错误处理 .....	75
将事务写入日志 .....	78
使用 XA 进行分布式事务处理 .....	83
二进制日志的组提交 .....	86
基于行的复制 .....	88
启用基于行的复制 .....	89
使用混合模式 .....	90
二进制日志管理 .....	90
二进制日志和系统崩溃安全 .....	91
binlog 文件轮换 .....	92
事故 .....	94



清除 binlog 文件 .....	94
mysqlbinlog 实用工具 .....	95
基本用法 .....	96
解释事件 .....	104
二进制日志的选项和变量 .....	108
基于行的复制参数 .....	110
小结 .....	111
<b>第 5 章 面向高可用性的复制 .....</b>	<b>112</b>
冗余 .....	113
计划 .....	114
slave 故障 .....	115
master 故障 .....	115
relay 故障 .....	116
灾难恢复 .....	116
方法 .....	116
热备份 .....	118
双主结构 .....	122
提升 slave .....	131
环形复制 .....	135
小结 .....	137
<b>第 6 章 面向横向扩展的 MySQL 复制 .....</b>	<b>138</b>
横向扩展读操作，而不是写操作 .....	140
异步复制的价值 .....	141
管理复制拓扑 .....	142
应用层的负载均衡 .....	145
级联复制 .....	153
配置 relay .....	154
使用 Python 添加 relay .....	155
专用 slave .....	156
过滤复制事件 .....	157
使用过滤将事件分配给 slave .....	159
数据的一致性管理 .....	160
非级联部署的一致性 .....	161
级联部署的一致性 .....	163

小结 .....	169
<b>第 7 章 数据分片 .....</b>	<b>171</b>
什么是数据分片 .....	172
为什么要分片 .....	173
分片的局限性 .....	174
分片方案的要素 .....	176
高级分片架构 .....	177
数据分区 .....	178
分配分片 .....	182
映射分片关键字 .....	186
分片方案 .....	186
分片映射函数 .....	190
处理查询和事务调度 .....	194
处理事务 .....	195
分配查询 .....	197
分片管理 .....	199
将分片迁移到其他节点 .....	199
分割分片 .....	203
小结 .....	203
<b>第 8 章 深入复制 .....</b>	<b>204</b>
复制架构基础 .....	205
中继日志的结构 .....	206
复制线程 .....	209
启动和停止 slave 线程 .....	210
通过 Internet 运行复制 .....	211
使用内置支持建立安全复制 .....	212
使用 Stunnel 建立安全复制 .....	213
细粒度控制复制 .....	215
关于复制状态的信息 .....	215
处理断开连接的选项 .....	223
slave 如何处理事件 .....	224
管理 I/O 线程 .....	224
SQL 线程的处理 .....	225
半同步复制 .....	231

配置半同步复制 .....	232
监控半同步复制 .....	234
全局事务标识符 .....	234
使用 GTID 配置复制 .....	235
使用 GTID 进行故障转移 .....	237
使用 GTID 提升 slave .....	238
GTID 的复制 .....	240
slave 的安全和恢复 .....	242
同步、事务以及数据库崩溃问题 .....	242
事务型复制 .....	244
保护非事务型语句的规则 .....	248
多源复制 .....	248
基于行的复制的细节 .....	251
Table_map 事件 .....	253
行事件的结构 .....	255
行事件的执行 .....	256
事件和触发器 .....	257
基于行的复制中的过滤 .....	259
部分行复制 .....	260
小结 .....	261
<b>第 9 章 MySQL 集群 .....</b>	<b>263</b>
什么是 MySQL 集群 .....	264
术语和组件 .....	264
MySQL 集群和 MySQL 有何不同 .....	265
典型配置 .....	265
MySQL 集群的特点 .....	266
本地和全局冗余 .....	268
日志处理 .....	268
冗余和分布式数据 .....	269
MySQL 集群的架构 .....	269
如何存储数据 .....	271
分区 .....	274
事务管理 .....	275
联机操作 .....	275
配置实例 .....	276

入门.....	277
启动 MySQL 集群.....	279
测试集群.....	283
关闭集群.....	284
获得高可用性.....	284
系统恢复.....	287
节点恢复.....	288
复制.....	289
获得高性能.....	293
高性能的注意事项.....	294
高性能的最佳实践.....	295
小结.....	297

## 第 2 部分 监控和管理

<b>第 10 章 监控入门.....</b>	<b>300</b>
监控方法.....	301
监控的好处.....	301
监控系统组件.....	302
处理器.....	302
内存.....	304
磁盘.....	304
网络子系统.....	306
监控方案.....	306
Linux 和 UNIX 监控.....	307
进程活动.....	308
内存利用率.....	312
磁盘利用率.....	314
网络活动.....	317
常见系统统计信息.....	318
使用 cron 自动监控.....	319
Mac OS X 监控.....	320
System Profiler.....	320
控制台.....	322
Activity Monitor.....	324
Microsoft Windows 监控.....	327



Windows 体验 .....	327
系统健康报告 .....	329
事件查看器 .....	331
可靠性监视器 .....	333
任务管理器 .....	334
性能监视器 .....	335
预防性维护监控 .....	337
小结 .....	337
<b>第 11 章 监控 MySQL .....</b>	<b>339</b>
什么是性能 .....	340
MySQL 服务器监控 .....	340
如何显示 MySQL 性能 .....	341
性能监控 .....	342
SQL 命令 .....	342
mysqladmin 实用工具 .....	348
MySQL 工作台 .....	350
第三方工具 .....	360
MySQL 基准测试套件 .....	362
服务器日志 .....	364
性能模式 .....	366
概念 .....	367
入门 .....	369
使用性能模式诊断性能问题 .....	377
MySQL 的监控分类 .....	378
数据库性能 .....	380
衡量数据库的性能 .....	380
数据库优化的最佳实践 .....	392
提高性能的最佳实践 .....	400
一切都很慢 .....	400
查询慢 .....	400
应用慢 .....	401
复制慢 .....	401
小结 .....	401

<b>第 12 章 监控存储引擎 .....</b>	<b>403</b>
InnoDB.....	403
使用 SHOW ENGINE 命令 .....	406
使用 InnoDB 监视器 .....	409
监控日志文件.....	413
监控缓冲池.....	414
监控表空间.....	416
使用 INFORMATION_SCHEMA 表 .....	417
使用 PERFORMANCE_SCHEMA 表.....	418
其他需要考虑的参数.....	419
InnoDB 故障排除的技巧 .....	420
MyISAM .....	422
优化磁盘存储.....	423
修复表.....	423
使用 MyISAM 实用工具 .....	424
按索引顺序存储表 .....	425
压缩表.....	426
对数据表进行碎片整理.....	426
监控 key cache.....	426
预加载 key cache.....	427
使用多个 key cache .....	428
其他需要考虑的参数.....	429
小结 .....	430
<b>第 13 章 监控复制 .....</b>	<b>432</b>
入门 .....	432
服务器设置 .....	433
包容性和排他性复制 .....	433
复制线程 .....	435
监控 master .....	437
master 的监控命令 .....	437
master 的状态变量 .....	441
监控 slave.....	441
slave 的监控命令 .....	442
slave 的状态变量.....	446
使用 MySQL 工作台监控复制 .....	447

其他需要考虑的问题 .....	449
网络 .....	449
监控和管理 slave 滞后 .....	450
slave 滞后的原因和预防措施 .....	450
使用 GTID .....	452
小结 .....	453
<b>第 14 章 复制的故障排除 .....</b>	<b>454</b>
哪里出错了 .....	455
master 上的问题 .....	455
master 崩溃及 Memory 表被占用 .....	455
master 崩溃及二进制日志事件丢失 .....	456
master 上查询正常但在 slave 上出错 .....	457
崩溃之后表损坏 .....	458
master 上的二进制日志损坏 .....	459
杀死非事务型表上长时间运行的查询 .....	459
不安全的语句 .....	460
slave 上的问题 .....	462
slave 服务器崩溃及复制无法启动 .....	462
slave 连接超时及反复重新连接 .....	463
slave 上的查询结果与 master 上的不同 .....	463
当尝试重启 SSL 时 slave 出错 .....	464
内存表数据丢失 .....	465
slave 崩溃后临时表丢失 .....	465
slave 运行慢而且与 master 不同步 .....	465
slave 崩溃后数据丢失 .....	466
崩溃后表损坏 .....	466
slave 上中继日志损坏 .....	467
slave 重启时的多个错误 .....	467
slave 上事务失败的后果 .....	467
I/O 线程的问题 .....	467
SQL 线程的问题：不一致 .....	468
slave 上的错误不一样 .....	468
高级复制问题 .....	469
变更没有在拓扑中复制 .....	469
环形复制的问题 .....	469

多 master 的问题 .....	470
HA_ERR_KEY_NOT_FOUND 错误 .....	470
GTID 问题 .....	470
复制的故障排除工具 .....	471
最佳实践 .....	472
了解你的拓扑结构 .....	472
检查所有服务器的状态 .....	475
检查日志 .....	475
检查配置 .....	475
有序地执行关闭操作 .....	475
有序地执行故障后的重启操作 .....	476
手动执行失败的查询 .....	476
不要混合使用事务型表和非事务型表 .....	477
一般步骤 .....	477
报告复制错误 .....	478
小结 .....	479
<b>第 15 章 保护你的资产 .....</b>	<b>481</b>
什么是信息保护 .....	482
信息保障的三个实践 .....	482
信息保障为什么重要 .....	483
信息完整性、灾难恢复及备份的职责 .....	483
高可用性与灾难恢复 .....	484
灾难恢复 .....	484
数据恢复的重要性 .....	489
备份和恢复 .....	490
备份实用程序和操作系统层的解决方案 .....	494
MySQL 企业备份 .....	495
使用 MySQL 实用工具集进行数据库的导出和导入 .....	507
mysqldump 工具 .....	507
物理文件复制 .....	510
逻辑卷管理器快照 .....	511
XtraBackup .....	516
备份方法的比较 .....	516
备份和 MySQL 复制 .....	517
使用复制进行备份和恢复 .....	518



PITR .....	518
自动备份 .....	526
小结 .....	528
<b>第 16 章 MySQL 企业版监控 .....</b>	<b>530</b>
MySQL 企业版监控入门 .....	531
产品 .....	532
剖析 MySQL 企业监控器 .....	532
安装概述 .....	533
MySQL 企业监控组件 .....	537
Dashboard .....	537
监控代理 .....	539
advisor .....	539
查询分析器 .....	541
MySQL 产品支持 .....	542
使用 MySQL 企业版监控 .....	542
监控 .....	544
查询分析器 .....	549
更多信息 .....	551
小结 .....	551
<b>第 17 章 使用 MySQL 实用工具管理 MySQL 复制 .....</b>	<b>553</b>
常见的 MySQL 复制任务 .....	554
状态检查 .....	554
停止复制 .....	557
添加 slave .....	558
MySQL 实用工具 .....	560
入门 .....	560
不通过工作台使用实用工具 .....	560
通过工作台使用实用工具 .....	560
常用工具 .....	562
比较数据库的一致性：mysqldbcompare .....	562
复制数据库：mysqldbcopy .....	565
导出数据库：mysqldbexport .....	566
导入数据库：mysqldbimport .....	569
发现不同：mysqldiff .....	570

显示磁盘使用情况：mysqldiskusage .....	574
检查表的索引：mysqlindexcheck .....	577
查找元数据：mysqlmetagrep .....	578
查找进程：mysqlprocgrep.....	579
克隆服务器：mysqlserverclone.....	581
显示服务器信息：mysqlserverinfo .....	583
克隆用户：mysqluserclone .....	584
实用工具客户端：mysqluc .....	585
复制的实用工具.....	586
配置复制：mysqlreplicate.....	586
检查复制的配置：mysqlrplcheck.....	588
显示拓扑结构：mysqlrplshow .....	591
高可用的实用工具 .....	592
概念.....	592
mysqlrpladmin.....	593
mysqlfailover.....	598
创建自己的实用工具 .....	606
MySQL 实用工具的结构.....	606
自定义工具的示例 .....	607
小结 .....	616
<b>附录 A 复制的提示和技巧 .....</b>	<b>617</b>
<b>附录 B 一个 GTID 的实现.....</b>	<b>634</b>
<b>索引.....</b>	<b>645</b>

---

# 前言

本书的作者们参与编写了部分 MySQL 组件，并在此领域工作了多年。Charles Bell 博士是带领 MySQL Utilities 小组的高级开发人员，同时还参与复制和备份工作。他的兴趣涵盖 MySQL 的各个方面，数据库理论、软件工程、微控制器和 3D 打印等。Mats Kindahl 博士是主要的高级开发人员，目前带领 MySQL 高可用性和扩展性小组，是若干 MySQL 特性的架构师和实现者。Lars Thalmann 博士是 MySQL Replication、Backup、Connectors 和 Utilities 小组的开发总监和技术领导，他设计了很多复制和备份的特性，主要从事 MySQL 集群、复制和备份技术的开发工作。

为了填补 MySQL 相关书籍的空白，我们撰写了这本书。关于 MySQL 有很多出色的书籍，但很少集中讲述它的高级特性和应用，诸如高可用性、可靠性和可维护性等。本书将涵盖所有这些主题，当然还有其他更多内容。

为了使阅读更加有趣，我们添加了一个遭遇老板提出种种要求的 MySQL 从业者的小故事。在该故事中，你将认识 Joel Thomas，最近他决定在一家刚开始使用 MySQL 的公司工作。你将看到 Joel 学习 MySQL 的方式，以及如何处理 MySQL 从业者所面临的一些最棘手的问题。希望你会觉得这部分内容很有趣。

## 读者对象

本书的读者对象是 MySQL 从业人士。我们假设读者已拥有 SQL、MySQL 管理和操作系统的基础背景知识。我们会介绍一些关于复制、灾难恢复、系统监控及其他以高可用性为主题的背景信息。其他书籍的第 1 章也会介绍相关有用的背景知识。

## 本书的组织结构

本书分为两部分。第 1 部分包括 MySQL 的高可用性和横向扩展性。由于这些问题很大程度上取决于复制，所以本部分大多都集中在这个主题上。第 2 部分介绍构建健壮的数据

据中心时，监控和性能方面的问题。

## 第 1 部分 高可用性和可扩展性

第 1 章 引言 介绍了本书的价值，并提供了阅读的情境。

第 2 章 MySQL Replicant 库 介绍了贯穿本书的一个 Python 库。

第 3 章 MySQL 复制原理 讨论了设置基本复制的手动和自动流程。

第 4 章 二进制日志 解释了与复制、灾难恢复、故障排除和其他管理任务相关的关键文件。

第 5 章 面向高可用性的复制 给出了服务器故障恢复的多种方法，包括自动化脚本的使用。

第 6 章 面向横向扩展的 MySQL 复制 介绍了提升大数据集读扩展性的多种技术和拓扑结构。

第 7 章 数据分片 描述了处理超大数据库的技术，以及通过分片提升数据库的写扩展性。

第 8 章 深入复制 讲述了诸如安全数据传输和基于行的复制等主题。

第 9 章 MySQL 集群 描述了如何使用该工具达到高可用性。

## 第 2 部分 监控和管理

第 10 章 监控入门 介绍了必须注意的主要操作系统参数，以及监控它们的工具。

第 11 章 监控 MySQL 介绍了几种数据库行为和性能的监控工具。

第 12 章 监控存储引擎 更加详细地解释了需要监控的参数，重点描述 MyISAM 或者 InnoDB 相关的问题。

第 13 章 监控复制 详细描述了如何跟踪主节点和从节点。

第 14 章 复制的故障排除 介绍了如何处理故障、重启、崩溃及其他意外事故。

第 15 章 保护你的资产 解释了备份和灾难恢复技术的使用。

第 16 章 MySQL 企业版监控 介绍了用于简化上述很多任务的一个工具套件。

第 17 章 使用 MySQL 实用工具管理 MySQL 复制 介绍了 MySQL 实用工具，包含管理 MySQL 服务器的一整套工具。



## 附录

附录 A 为复制的提示和技巧，列举了特定情况下一些有用的方法。

附录 B 为一个 GTID 的实现，给出了如果服务器不支持 GTID，处理事务故障转移的实现方法。

## 本书的印刷约定

下面是本书中使用的字体约定：

纯文本 (Plain text)

表示菜单标题、选项和按钮。

斜体 (*Italic*)

表示新术语、表名和数据库名、URL、E-mail 地址、文件名及 UNIX 工具。

等宽字体 (Constant width)

表示命令行选项、变量和其他代码元素、文件内容及命令输出。

等宽加粗字体 (**Constant width bold**)

表示命令或其他应该由用户输入的文本。

等宽斜体 (*Constant width italic*)

表示应该替换为用户提供的值的文本。




这个图标表示提示或建议。



这个图标表示一般说明。



这个图标表示警告或注意。

中文版书中切口以“”表示原书页码，便于读者与原英文版图书对照阅读，本书的索引中所列的页码为原英文版图书中的页码。

## 使用示例代码

补充材料（包括代码示例、练习等）可以到 <http://bit.ly/mysqllaunch> 下载。

本书对你的工作有所帮助。一般来说，可以在你的程序或者文档中使用本书提供的示例代码。你不必联系我们获得许可，除非你要大量传播代码。例如，从书中抄几块代码编写程序不需要许可；销售或分销 O'Reilly 随书附带光盘上的示例代码则需要许可；引用本书的示例代码回答问题不需要许可；将书中大量的示例代码附加到你的产品文档中则需要许可。

我们感谢但不要求注明出处。出处的格式一般包括标题、作者、出版商和 ISBN。例如，“*MySQL High Availability*, by Charles Bell, Mats Kindahl, and Lars Thalmann. Copyright 2014 Charles Bell, Mats Kindahl, and Lars Thalmann, 978-1-44933-958-6.”

如果你觉得示例代码的使用不合理或不符合以上的许可权限，请随时联系我们：  
[permissions@oreilly.com](mailto:permissions@oreilly.com)。

## Safari Books Online



Safari Books Online ([www.safaribooksonline.com](http://www.safaribooksonline.com)) 是一个按需出版的数字图书馆，出版各种专业的书籍和视频，它们由世界上技术和商业领域的优秀作者撰写。

技术人员、软件开发、网页设计者及商业和创意专业人士都把 Safari Books Online 当作科研、问题解决、学习及认证训练的主要资源。

Safari Books Online 为组织、政府机构和个人提供了大量的产品组合和定价方案。订阅者可以从一个完全可搜索的数据库中获取成千上万的书籍、培训视频和尚未出版的手稿，这些出版商包括 O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology，以及其他数十家出版社。如想了解更多 Safari Books Online 的信息，请在线访问我们。

# 如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者：

美国：

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）  
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网站，你可以在那里找到关于本书的相关信息，包括勘误列表、示例代码以及其他信息。本书的网站地址是：

[http://bit.ly/mysql\\_high\\_availability](http://bit.ly/mysql_high_availability)

对于本书的评论和技术性的问题，请发送电子邮件到：

[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)

关于我们的书籍、课程、会议和新闻的更多信息，请参阅我们的网站 <http://www.oreilly.com>。

在 Facebook 上找到我们：<http://facebook.com/oreilly>

在 Twitter 上关注我们：<http://twitter.com/oreillymedia>

在 YouTube 上观看我们：<http://www.youtube.com/oreillymedia>

## 致谢

作者要感谢这一版和上一版的技术审校人员：Mark Callaghan, Morgan Tocker, Sveta Smirnova, Luis Soares, Sheeri Kritzer Cabral, Alfie John 和 Colin Charles。你们对细节的关注以及有见地的建议都是无价的。没有你们的帮助，就没有高质量的图书。

还要感谢我们 MySQL 小组极有才华的同事们，以及 MySQL 社区提供评论的人们，包括 Alfranio Correia, Andrei Elkin, Zhen-Xing He, Serge Kozlov, Sven Sandberg, Luis Soares, Rafal Somla, Li-Bing Song, Ingo Strüwing, Dao-Gang Qu, Giuseppe Maxia 和 Narayanan Venkateswaran，他们孜孜不倦的努力使得 MySQL 变得健壮和强大。特别感

谢 MySQL 客户支持的专家们，他们帮助我们缩小了客户需求与我们改进产品的愿望之间的差距。还要感谢很多社区成员，他们如此忘我地投入时间和精力改善 MySQL，造福大家。

最后，同时也是最重要的，要感谢我们的编辑，Andy Oram，他帮助我们完成这项工作，并忍受着我们对于 MySQL 时而理智时而过度积极的热情。向整个 O'Reilly 小组致以最诚挚的谢意，特别是编辑，感谢他们的耐心，我们如此努力地往一本已经很庞大的书里面塞了这么多新的内容。

Charles 要感谢他最爱的妻子，Annette，当他忙于本书的工作而不在家的时候，感谢她的耐心和理解。Charles 还要感谢他在 Oracle 工作的 MySQL 小组的同事们，他们每天无偿地将自己的智慧贡献给每个人。最后，Charles 要感谢他基督教的兄弟姐妹们，他们每天都在考验和支持着他。

Mats 要感谢他的妻子 Lill 和两个儿子 Jon 和 Hannes，感谢他们在自己最困难的时候给予无条件的爱和理解。他们是他一生的挚爱，他无法想象没有他们的生活。Mats 还要感谢他在 Oracle 内外的 MySQL 同事们，以及所有那些有趣、惊喜和鼓舞人心的时光：你们是这行中最能干的人。

Lars 要感谢他的女朋友 Claudia，他爱她溢于言表。他还要感谢现在和以前的所有同事，他们让 MySQL 成为一个十分有趣的工作场所。事实上，这并不是一个场所。MySQL 开发小组的分布式本质和很多专业开发者的开放胸怀是真正了不起的。MySQL 社区有一种特殊的精神，使 MySQL 工作成为一个光荣的任务。我们共同创造着非凡。令人吃惊的是，最初的这样一小撮人，成功地创造了一个今天服务于许多财富 500 强公司的产品。



# 高可用性和可扩展性

数据库为应用程序提供高可用性和可扩展性的重要特性之一就是复制（Replication）。复制在数据库层创建冗余，同时产生多个数据库副本进行读扩展。第 1 部分讲述了怎样使用复制达到高可用性，以及如何扩展你的系统。

# 引言

为了找份新工作，Joel 仔细浏览了分类广告。虽然目前他的工作很好，而且自从他上大学以来，公司就待他不薄。但是已经毕业这么多年了，他希望在职业生涯中能有更多的挑战。

“这个看上去不错。”他边说边在一个招聘 MySQL 计算机专家的广告上画了个圈。他有 MySQL 经验，也符合这份工作的学历要求。在快速浏览了其他几个广告后，他决定电话咨询这个 MySQL 的工作。大致问了几个问题之后，人力资源部经理让他两天后去面试。

经过两天三轮的面试以后，他被引见给公司的董事长兼首席执行官 Robert Summerson，进行最后一轮的技术面试。提问过程中，Summerson 先生停下来参看笔记，Joel 在旁边等着。到目前为止，大多是些关于信息技术的普通问题，但 Joel 知道关于 MySQL 的刁难问题还在后面。

终于，面试官说：“Thomas 先生，我对你的回答印象很深刻。我可以叫你 Joel 吗？”

“可以，先生。”Joel 说，这时面试官第三次看了笔记，Joel 又是一阵不自在。

“说说你对 MySQL 的了解吧。”Summerson 先生把手放在桌子上，凝视着 Joel。

Joel 开始解释他所知道的 MySQL，倒腾着前一天晚上看的一大堆资料。十分钟后他把要说的都说完了。

过了几分钟，Summerson 先生站起来，并向 Joel 伸出一只手。当 Joel 起身与 Summerson 先生握手时，Summerson 说：“这就是我想听的东西，Joel。这份工作是你的了。”

“谢谢您，先生。”

Summerson 先生示意 Joel 随他一起走出办公室。“我带你去人力资源部，这样我们可以把你加到公司员工的工资名单上。从周一开始先试用两周可以吗？”

Joel 非常高兴，不由得笑了：“好的，先生。”

“很好。” Summerson 先生再次与 Joel 握手，说道：“我希望你来的时候已经做好评估我们的 MySQL 服务器配置的准备。我需要一份关于服务器配置和健康状况的完整报告。”

Joel 开车出停车位的时候，从之前的兴奋中冷静了下来。他并没有立即回家，而是去了最近的书店。“我需要一本 MySQL 的好书”，他想。

现在，你决定要进行大规模安装和维护工作。好吧，你将会迎来一段非常有趣且有益的时光。

与运行一个小网站相比，支撑一个大企业需要计划、远见、经验，甚至更周密的规划。作为一个大型企业的数据库管理员，你需要或者即将需要做以下的事情：

- 为核心业务数据提供灾难恢复计划。这个过程可能需要执行不止一次。
- 计划管理大量用户库，监控各节点的负载，并提供优化方案。
- 当用户数量急剧增长时，准备好快速横向扩展计划。

对于上述情况，必须提前计划并准备好快速应对方案。

并不是所有大量使用服务器的应用都是网站，因此我们使用“部署 (deployment)”这个术语，而不是站点 (site) 或网站 (website)，来表示用于支持某种应用的服务器。它可能是一个网站，也可能只是一个 CRM (客户关系管理) 系统或者一个在线游戏。本书侧重于该系统的数据库层，但也有一些应用层和数据库层的整合例子。

为了保证站点可响应和可用，你需要三样东西：数据备份 (backup)、冗余 (redundancy) 和响应性 (responsiveness)。备份可以将节点恢复到它崩溃之前的状态；冗余保证即使在一个或更多的节点停止提供服务的情况下，站点仍能继续运行；响应性保证系统在实际生产中可用。

执行备份有多种方法，方法的选择取决于你的需求。你需要即时恢复到一个精确的时间点吗？如果是，就必须满足执行基于时间点恢复 (point-in-time-recovery, PITR) 必需的条件。你想在备份的同时保持服务器正常运行吗？如果是，就必须保证使用的备份方

法不会扰乱运行中的服务器，比如在线备份。<sup>注1</sup>

冗余是通过硬件副本来实现的，让几个实例同时运行，并通过复制在几个机器上保存相同数据的多个可用副本。如果其中一个机器失效，可以切换到另一个拥有相同数据副本的机器。

和复制一样，备份也在系统扩展和添加新节点方面起了重要作用。打个比方，如果使用正确，甚至可能按下按钮就可以自动添加新的从节点。

## 到底什么是复制

阅读本书时你可能已经对复制有所了解。尽管如此，这里还是要回顾一下概念和思想。

复制就是把某个服务器上（称为主节点服务器或者简称主节点，即 master）的所有变化克隆到另一个服务器（称为从节点服务器或简称从节点，即 slave）。复制通常用来创建 master 的一个可靠副本，不过复制也有其他用途。

两种最常见的使用复制的例子是：（1）创建一个 master 的备份，以避免 master 崩溃时丢失数据；（2）创建一个 master 的副本，从而在不干扰其他业务的情况下执行报表和分析工作。

对小型企业来说，这足以简化很多事情，但复制可以做得更多，比如：

支持多个机房

每个地点都可能要维护服务器，然后将变更复制到其他地方，从而使得信息处处可用。这就需要保护数据，在合法的情况下，保证用于审计目的的业务信息可用。

6 即使有服务器停机，也能保证业务的持续运行

如果原来的服务器失效，由其他服务器处理所有的访问量。

即使有灾难发生，也能保证业务持续运行

复制可以将变更发送给不同地理位置的其他数据中心。

错误保护（oops）

当 slave 连接 master 时，可从总是比 master 落后一个固定的时间周期（例如 1 小时），这样就会产生一个延迟的 slave（Delayed Slave）。如果这时 master 上发生错误，可以找到出错的语句，然后在 slave 执行之前删除它。

---

注1 不一定要用单一备份方法，也可以根据需要混合使用多种不同的方法。但无论如何，你都要选择最适合的备份方案。



目前很多应用程序中使用复制最重要的场景之一就是横向扩展（scale out）。现今的应用程序通常是读密集型的，具有高读写比。为了减少 master 上的负载，你可以搭建一个专门响应读请求的 slave。通过一个负载均衡器，可以将读请求定向到合适的 slave，而写请求则交给 master 处理。

在横向扩展的场景下使用复制时，理解 MySQL 复制的异步性（asynchronous）很重要，即事务首先在 master 上提交，然后复制到 slave 并在 slave 上执行。这意味着 master 和 slave 可能并不一致，而且如果复制持续运行，slave 将会落后于 master。<sup>注2</sup>

使用异步复制的好处在于它比同步复制更快、更具可扩展性，但在那些实时数据很重要的情况下，必须处理好不同步的问题以保证信息的时效性。

然而，读扩展并不足以适用于所有应用程序。随着需求增加，数据库更大，写负载更高，需要扩展的就不只是读操作了。管理大型数据库，提升大型数据库系统的性能，可以通过分片（sharding）技术来实现。通过分片，可以将数据库划分为若干可管理的数据分片，将数据库分发到多个服务器上，从而增加数据库的规模，并有效地扩展写操作。

复制的另一个重要应用是通过添加冗余来保证高可用性。最常见的技术是使用双主配置（dual-master），即通过复制保持一对 master 总是可用，其中每个 master 都是对方的镜像。如果其中一个 master 失效，另一个会立即接手。

除了双主配置，还有其他与复制无关的高可用性技术，如使用共享存储或复制磁盘。尽管它们不是 MySQL 特有的，但这些技术对于保证高可用性来说也是很重要的工具。

◀ 7

## 那么，是否需要备份

备份策略是保持系统可用的一个关键部分。常规的服务器备份提供了应对崩溃和灾难的安全措施，它们在某种程度上可以通过复制来实现。然而，即使正确并高效地使用了复制技术，还是有一些复制无法解决的问题。你需要有一个切实可行的备份策略来应对以下情况。

### 错误保护

如果某个错误在它实际发生以后很长时间才被发现，这时复制便不再有效。在这种情况下，必须把系统回滚到错误发生前的时刻并解决问题。这需要一个切实可行的备份计划。

---

注2 还有一个扩展叫作半同步复制（semisynchronous replication），参见第8章的“半同步复制”小节。直到版本 MySQL 5.7.2 DMR，在复制之前就将事务外部化，允许复制前读取，在高可用性应用时需要注意。

如果使用的是有延迟的 slave，复制可以提供一些错误保护；但如果错误在延迟时间段之后才被发现，这些变更已经在 slave 上生效了。所以，一般来说，仅仅使用复制不可能做到完全的错误保护，还需要备份。

#### 创建新服务器

无论用于横向扩展的 slave，还是备用的新 master，创建新服务器都需要对已有服务器做备份，并在新服务器上恢复这个备份映像。这需要有一个快速高效的备份方法来最小化宕机时间，并保持系统负载维持在一个可接受的水平。

#### 法律原因

除了纯粹业务原因需要保护数据外，法律规定也可能要求保证数据安全，即使在灾难发生时。不遵守这些规定会给业务运作带来重大问题。

简而言之，不管有没有其他的预防措施来保证数据的安全，备份策略对于业务运作都是必需的。

## 什么是监控

即便已经正确搭建了复制，还有必要理解你的系统负载，并密切监控可能发生的任何问题。客户使用模式的改变将导致业务需求变化，需要平衡系统以尽可能高效地使用资源，降低由于资源利用的突然变更导致系统不可用性的风险。

8 为了应对这些变更，有很多监控、度量和计划的方法，比如：

- 为频繁读取的表添加索引。
- 重写查询或者改变数据库的结构，以缩短执行时间。
- 如果锁被长时间占用，表示多个连接正在使用同一个表，可能要切换存储引擎。
- 在横向扩展的数据库复制架构下，如果某些 slave 处理了大量的查询，处于过热状态，系统可能需要重新均衡，以保证所有 slave 都被平均地访问。
- 在处理资源使用的突然变更时，首先确定每个服务器的正常负载，然后了解在负载突然增加时，系统响应什么时候开始变慢。

如果不监控，就没有办法观察到有问题的查询、过热的 slave 或者使用不恰当的数据表等。

## 其他阅读材料

大量的文献讲述如何使用 MySQL 完成各种各样的工作，同样也有许多关于高可用性系统的著作。如果你打算从事 MySQL 工作，下面是我们强烈推荐的书籍清单：

《高性能 MySQL（第 3 版）》（电子工业出版社）

这是企业级 MySQL 应用的最佳书籍之一，包括如何优化查询，以及保证系统的响应性和可用性。

*Scalable Internet Architectures*，由 Theo Schlossnagle 编写（Sams Publishing）

作者是工业界最杰出的思想家之一，该书是从事系统扩展人员的必读本。

*MySQL*，由 Paul DuBois 编写（Addison-Wesley）

这是 MySQL 的参考书，一共 1200 页（真的！），涵盖了一切关于 MySQL 你想知道的内容（可能也有许多你不知道的）。

本书使用作者们开发的一个 Python 库（名为 *MySQL Python Replicant*）实现许多管理性任务。MySQL Python Replicant 可以从 <https://launchpad.net/mysql-replicant-python> 下载。

## 小结

9

下一章我们将开始介绍复制的基础知识，所以找个舒服的凳子，打开电脑，然后开始吧。

Joel 正在调整椅子，这时有人敲门。

“还习惯吗，Joel？” Summerson 先生问。

Joel 不知道该说些什么。第一天上班他就接到任务要搭建一个用于复制的 slave，尽管他花的时间比预期要长得多，但还得看老板怎么说。Joel 首先想到的回答是：“是的，先生，我还在倒腾这椅子。”

“文档写得不错，Joel。我想让你写一份报告，汇报一下你认为应该如何改善我们的数据库服务器管理。”

Joel 点点头：“没问题。”

“好。我再多给你一天的时间整理办公室。我希望周三下班之前看到报告，没事了。”

Joel 还没来得及回答，Summerson 先生就走开了。

Joel 坐下来，转了一下椅子的另一个手柄。“咔嚓”一声，靠背掉了，迫使他甩开双臂。“哇！”他看了一下门口，笨拙地捡起椅子，还好没有人看到他的即兴体操。“好了，这下手柄彻底坏了。”他说。

# MySQL Replicant库

Joel 打开他记满常用命令的文本文件，复制到另一个编辑器里，并根据需要做修改。这是一堆涉及大量工具和实用程序的命令。“啊，这工作太烦琐了！”他心想，“肯定有其他更好的办法。”

他抓狂地翻开手边厚重的《高可用 MySQL》，浏览了一下目录。“啊哈！有一章是关于复制的库，嗯，这正是我所需要的！”

对于大型部署来说，将管理过程自动化很重要。那么，你可能会问，“如果把本书里面的步骤都自动化了不是很好？”值得高兴的是，很多时候我们的确可以这样。这一章我们介绍 MySQL Replicant 库，这是由作者们开发的用于管理复制的简单库。我们将介绍一些基本原理和类，并在后续章节向这个库扩展新功能。

代码可以到 Launchpad (<http://bit.ly/mysqllaunch>) 下载，那里还有很多其他的源代码和文档可供下载。

Replicant 库的基本思想是建立一个服务器间的连接模型（可以使用任何计算机，比如笔记本电脑），如图 2-1 所示。设计这个库，可以通过更改模型来管理连接。例如，如果想要 slave 重新连接到其他 master，只要把模型中的 slave 重新连接就可以了，这个库会发送适当的命令完成这个工作。

除了图 2-1 所示的简单复制拓扑以外，还有两种基本拓扑，即树形拓扑和双主拓扑（用于提供高可用性）。关于拓扑的内容第 6 章会详细讲解。

为了让这个库适用于大多数平台和部署，它的构建遵循以下几点。

- 服务器可能运行在不同的操作系统上，比如 Windows、Linux 或者诸如 Solaris 和 Mac OS X 这样的类 UNIX 系统。根据操作系统的不同，服务器的启动和停止，以及配置文件的命名都有所不同。所以，这个库应该支持不同的操作系统，并且能够扩展到库中没有的新操作系统。



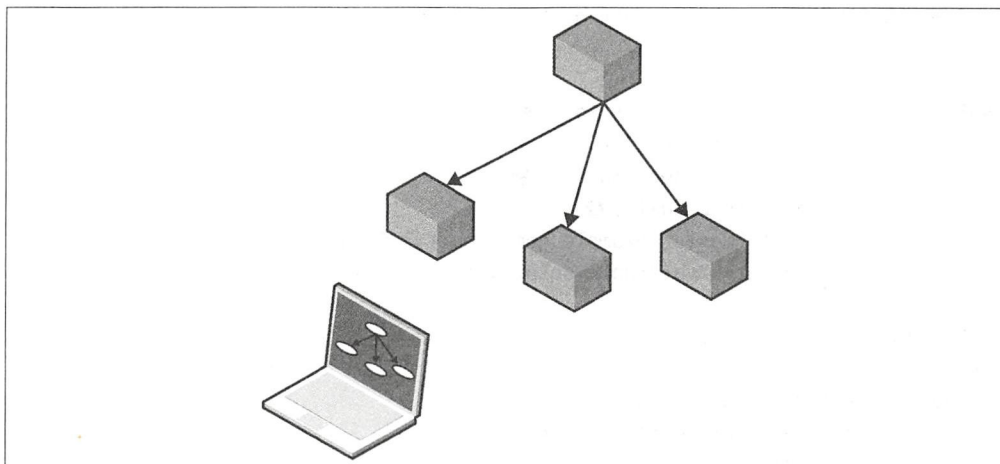


图2-1：模型的一种复制拓扑

- 部署可能包含多个不同版本的 MySQL。例如，将部署升级到新版本的时候，可能存在新旧版本混合的情况。库应该能够应付这种部署。
- 部署中的服务器有很多不同的角色，所以可能需要为服务器指定不同的角色。而且应该尽可能创建好那些一开始无法预期的新角色。另外，服务器应该能够更改角色。
- 每个服务器都要能够执行 SQL 查询。在配置和提取部署管理信息的时候，需要这个功能。另外，系统其他部件执行任务的时候也需要这个功能，例如 slave 提升的实现。
- 每台机器都要能够执行 shell 命令。有些管理性任务 SQL 接口无法完成，就需要用 shell 命令。此外，还可用于操作系统部分管理服务器。
- 能够添加或删除服务器配置文件的选项。
- 这个库要支持单台机器上有多个服务器的部署情况。这就需要区分单台机器上不同 MySQL 服务器的部署文件和数据库文件。
- 应该提供一组执行常用任务的实用程序，例如搭建复制。同时，它还要能够扩展那些一开始不支持的新功能。

13

这个接口尽可能多地隐藏了复杂度，并以一个简单的 Python 接口展现。作者们选择 Python 是因为它简捷、易读，并且支持所有运行 MySQL 的操作系统，在通用脚本方面越来越受欢迎。示例 2-1 给出了一个定义拓扑的例子。

#### 示例2-1：使用库创建拓扑

```
from mysql.replicant.server import Server, User
from mysql.replicant.machine import Linux
```

```

from mysql.replicant.roles import Master, Final

# 我们使用的 master
MASTER = Server('master', server_id=1, ❶
                sql_user=User("mysql_replicant"), ❷
                ssh_user=User("mats"), ❸
                machine=Linux(), ❹
                host="master.example.com", port=3307, ❺
                socket='/var/run/mysqld/mysqld.sock')

```

```

# 可用的 slave
SLAVES = [ ❻
    Server('slave1', server_id=2,
          sql_user=User("mysql_replicant"),
          ssh_user=User("mats"),
          machine=Linux(),
          host="slave1.example.com", port=3308),
    Server('slave2', server_id=3,
          sql_user=User("mysql_replicant"),
          ssh_user=User("mats"),
          machine=Linux(),
          host="slave2.example.com", port=3309),
    Server('slave3', server_id=4,
          sql_user=User("mysql_replicant"),
          ssh_user=User("mats"),
          machine=Linux(),
          host="slave3.example.com", port=3310),

```

14

```

]

```

```

# 为这些服务器创建角色
master_role = Master(User("repl_user", "xyzyzy")) ❼
slave_role = Final(MASTER) ❽

```

```

# 为服务器注入角色
master_role.imbue(MASTER) ❾
for slave in SLAVES:
    slave_role.imbue(slave)

```

```

# 表示所有服务器的变量
SERVERS = [MASTER] + SLAVES

```

- ❶ 第一步创建一个服务器对象，包含访问服务器所需的所有信息。这个服务器用作 master，但这个语句并没有做什么特别的设置，当调用 imbue 方法注册一个角色的时候结束。

- ❷ 配置服务器的时候，你需要知道如何连接服务器以及如何向它发送 SQL 命令。在这个例子中，我们有一个专门的 replicant 用户用来访问服务器。这里没有密码，当然你也可以在创建 User 接口的时候设置密码。
- ❸ 还需要访问运行服务器的机器，以便停止服务器或者访问配置文件等。这一行授权用户连接到这个机器。
- ❹ 由于不同操作系统启动和停止服务器的方式不同，必须指定服务器运行在什么操作系统上。这里所有的服务器都运行在 Linux 上。
- ❺ 这是服务器所在的主机信息。host 和 port 用于连接远程机器，socket 用于连接同一台机器。如果仅仅是远程连接，可以省略 socket。
- ❻ 创建一组 slave 服务器。提供用于连接 master 的基本信息，但不涉及 slave 的具体参数。
- ❼ 调用 imbue 方法为服务器配置角色。该语句创建一个 Master 角色，其中复制用户是 slave 连接服务器用的。
- ❽ 将 slave 角色声明为 Final，这样不会在服务器上执行二进制日志，所以以后无法被提升为 master。
- ❾ 这些语句为所有服务器调用 imbue 方法设置角色，这会更新每个服务器的配置，然后应用到那个角色中去。必要的话（例如不得不更改配置文件的话），这些语句还会重启服务器。

◀ 15

上一个例子为所有服务器都赋予了角色。启动所有服务器之后，示例 2-2 告诉你怎样使用库将所有 slave 重定向到新的 master。

示例2-2：使用库重定向slave

```
import my_deployment

from mysql.replicant.commands import change_master

for slave in my_deployment.slaves:
    slave.stop()
    change_master(slave, my_deployment.master)
    slave.start()
```

我们故意将这个例子简化，所以省略了某些重要步骤。如代码所示，立即停止复制，如果事务在动态服务器上执行的话可能会导致事务丢失。第 5 章描述了如何正确地更换 master。

后续章节会介绍如何实现这些应用。为了避免不必要的混乱代码，我们去掉了一些错误检查和其他稳定及安全等防护性措施。想要得到这个库的完整代码，请到 Launchpad 上下载 (<http://bit.ly/mysqllaunch>)。

## 基本类和函数

要想使用这个库，首先需要了解一些常用概念的基本定义，例如用异常来报告错误，用简单对象来表示位置和用户信息。

库中给出了完整的异常列表。所有异常都继承自同一个基类 `Error`，当然按照惯例，库中也有定义。涉及的异常有如下一些。

### `EmptyRowError`

当 `select` 查询不返回任何行的时候抛出这个异常。

### `NoOptionError`

当 `ConfigManager` 找不到配置选项的时候抛出这个异常。

### 16 `SlaveNotRunningError`

当 `slave` 应该运行但却没有运行的时候抛出这个异常。

### `NotMasterError`

当服务器不是 `master` 并且操作非法的时候抛出这个异常。

### `NotSlaveError`

当服务器不是 `slave` 并且操作非法的时候抛出这个异常。

还有一些书中用到的表示常用概念的类，如下所示。

### `Position` 和 `GTID`

这些类表示 `binlog` 位置，包括文件名和文件内的字节偏移，或者全局事务标识符 (`GTID`, MySQL 5.6 引入)。使用表示方法将 `binlog` 位置以一种可解析的表示形式输出出来，因而 `binlog` 位置可以存在二级存储中，或者在你需要查看的时候才用。

为了对这些位置进行比较和排序，该类定义了一个比较运算符对位置进行排序。

注意，如果不采用全局事务标识符，那么不同服务器上的位置可能有所区别，所以比较不同服务器上的位置是没有用的。因此，如果试图比较不同类型的位置，将会抛出异常。



## User

这个类代表用户，包括用户名和密码，可用于多种类型的账号，比如 MySQL 用户账号、shell 用户账号、复制用户（稍后介绍）等。

# 对各种操作系统的支持

为了支持不同的操作系统，你可以用一组类将它们之间的区别抽象出去。基本思想是为不同操作系统上的任务实现其各自的类方法。这样，我们只需要这些方法就可以停止和启动服务器了。

## Machine

这个类是机器的基类，存储机器的共同信息。一个机器实例至少包括以下成员：

`Machine.defaults_file`

该机器上 *my.cnf* 文件的默认存储位置

`Machine.start_server(server)`

启动服务器的方法

`Machine.stop_server(server)`

停止服务器的方法

◀ 17

## Linux

这个类负责处理运行在 Linux 机器上的服务器，通过存储在 */etc/init.d* 位置的 *init(8)* 脚本来启动和停止服务器。

## Solaris

这个类负责处理运行在 Solaris 机器上的服务器，使用 *svadm(1M)* 命令来启动和停止服务器。

# 服务器

`Server` 类定义了接口中高级函数实现所需的所有基本功能，包括如下几项。

`Server(name, ...)`

`Server` 类表示系统中的一个服务器，整个系统中的每一个服务器都对应一个对象。这里是一些重要的参数（完整的参数列表请参考 Launchpad 上的项目页面）：

*Name*

这是服务器的名字，在创建 *pid-file*、*log-bin* 和 *log-bin-index* 选项的值时也

会用到。如果没有指定 `name` 参数，就从 `pid-file`、`log-bin` 和 `log-bin-index` 选项中推断其值；如果都没有指定，则使用默认值。

#### *host*、*port* 和 *socket*

`host` 是服务器所在的主机地址，`port` 是 MySQL 客户端连接服务器用的端口，`socket` 则用于同一台主机上的连接。

#### *ssh\_user* 和 *sql\_user*

使用用户和密码的组合连接机器或服务器。这些用户用来执行管理性命令，例如启动和停止服务器、读写配置文件，或者在服务器上执行 SQL 命令。

#### *machine*

存储操作系统特定原语的对象。我们用“`machine`”而不是“`os`”来命名这个对象，是为了避免与 Python 标准库模板 `os` 命名冲突。这个参数允许使用不同方法来启动和停止服务器或者执行其他任务。还有其他操作系统特定的参数，我们稍后再介绍。

18

#### *server\_id*

存储服务器标识的可选参数，在每个服务器的配置文件中定义。如果没有这个选项，那么服务器标识符将从配置文件中读取；如果配置文件也没有，那么这个服务器就不参与复制架构，既不是 `master` 也不是 `slave`。

#### *config\_manager*

这个选项存储到配置管理器的引用。从这个选项可以查询到服务器配置的相关信息。

### `Server.connect()` 和 `Server.disconnect()`

在会话中执行命令之前使用 `connect` 方法连接服务器，然后在结束会话之后使用 `disconnect` 方法断开连接。

在某些情况下，即使 SQL 命令执行结束还需要保持服务器连接有效，这时这两个方法就很有用。举例来说，执行 `FLUSH TABLES WITH READ LOCK` 的时候，如果连接失效了，锁就会自动释放。

### `Server.ssh(command, args...)` 和 `Server.sql(command, args...)`

用于在服务器上执行 shell 命令或 SQL 命令。

`ssh` 和 `sql` 方法都返回一个迭代类型，`ssh` 返回命令执行结果行的 `list`，而 `sql` 返回内部类 `Row` 对象的 `list`。`Row` 类定义 `__iter__` 和 `next` 方法迭代结果行，例如：

```
for row in server.sql("SHOW DATABASES"):
    print row["Database"]
```

为了处理返回单个行的语句，这个类还定义了 `__getitem__` 方法，用于读取单个行中的某个字段，如果没有行则抛出异常。也就是说，如果你知道返回结果只有一行（很多 SQL 语句都只返回一行），你就可以避免在刚才的例子中使用循环，像这样：

```
print server.sql("SHOW MASTER STATUS")["Position"]
```

`Server.fetch_config()` 和 `Server.replace_config()`

`fetch_config` 和 `replace_config` 方法将远程服务器中的配置文件读到内存中，这样用户可以增删配置选项或者更改某些选项的值。例如，用下面的模块可以添加 `log-bin` 和 `log-bin-index` 选项：

```
from my_deployment import master

config = master.fetch_config()
config.set('log-bin', 'capulet-bin')
config.set('log-bin-index', 'capulet-bin.index')
master.replace_config(config)
```

19

`Server.start()` 和 `Server.stop()`

`start` 和 `stop` 方法向 `machine` 对象发送相应信息，取决于服务器使用的操作系统。这两个方法分别完成启动和停止服务器的功能。

## 服务器角色

根据角色的不同，服务器的工作方法也略有不同。例如，`master` 要求 `slave` 通过复制用户来连接而 `slave` 不需要用户账号，除非 `slave` 同时担任 `master` 角色并且有其他 `slave` 请求连接。为了灵活地获得服务器配置，引入一些代表不同角色的类。

在服务器上使用 `imbue` 方法时，服务器将接收合适的命令去正确配置那个角色。注意，在整个部署周期中服务器可能变换角色，所以这里赋予的角色只是用于最初的部署配置。在部署中，服务器总有一个指定的角色，所以服务器还有一个关联角色。

如果服务器角色改变，可能需从服务器删除某些配置。因此还需要为角色定义一个 `unimbue` 方法，在服务器转换角色的时候使用。

本例中，只定义了三个角色。后面我们会定义更多其他角色，比如，在下面的例子中，我们将讲述如何创建用作备用服务器或中继服务器的 `nonfinal` 类型的 `slave`。MySQL Replicant 库中定义了以下三种角色。

## Role

这是所有角色的基类。每个派生类都需要定义 `imbue` 方法和 `unimbue` 方法（可选），为服务器赋予角色。为了帮助派生类完成一些常见任务，`Role` 类还定义了很多帮助函数，包括：

`Role.imbue(server)`

通过执行适当的代码，该方法为服务器赋予新的角色。

`Role.unimbue(server)`

在赋予其他角色之前，这个方法进行一些清理操作。

20

`Role._set_server_id(server, config)`

如果配置中没有服务器标识符，那么这个方法会将其设为 `server.server_id`；如果配置文件中服务器标识符，那么这个值会用来设置 `server.server_id` 的值。

`Role._create_repl_user(server, user)`

该方法在服务器上创建一个复制用户，并向它授予必要的权限，担任复制 `slave` 的角色。

`Role._enable_binlog(server, config)`

通过将 `log-bin` 和 `log-bin-index` 设置为适当的值，该方法在服务器上启用二进制日志。如果 `log-bin` 已经被设置过了，那么这个方法什么都不做。

`Role._disable_binlog(server, config)`

通过清空配置文件中 `log-bin` 和 `log-bin-index` 的值，该方法禁用二进制日志。

## Vagabond

这是所有服务器的默认角色，不参与复制部署。因此，这种 `vagabond` 服务器不承担任何职责。

## Master

这是 `master` 服务器的角色。这个角色会设置服务器标识符，启用二进制日志，并为 `slave` 创建复制用户。复制用户的用户名和密码将存储在服务器上，当 `slave` 请求连接的时候，这个类会查找复制用户名。

## Final

这是 `final` 类型 `slave`（即没有二进制日志的 `slave`）的角色。如果服务器被赋予这个角色，将会获得服务器标识符，二进制日志被禁用，而且连接 `master` 的时候使用 `CHANGE MASTER` 命令。

注意，在写回配置文件之前需要停止服务器，并在写完之后重新启动服务器。当服务器在运行的时候，配置文件是只读的，并且读完即关闭。为了保证文件的安全性，在修改它之前需要先停止服务器。

一个重要的设计理念是，角色不存储任何与服务器相关的信息。因此，要得到一份完整的 master 列表，需要向 role 对象添加 master；但由于服务器的角色在整个部署周期中是随时间变化的，所以仅在系统搭建的时候使用角色。由于角色可以包含参数，所以同样的信息可用于配置多台服务器。

```
import my_deployment

from mysql.replicant.roles import Final

slave_role = Final(master=my_deployment.master)
for slave in my_deployment.slaves:
    slave_role.imbue(slave)
```

21

## 小结

本章我们介绍了如何创建库来简化服务器的管理，并简单介绍了我们开发的 MySQL Replicant 库。

Joel 完成了脚本测试。他挺自信所有代码都没问题，最后的命令应该能节省他以后不少时间。他按下了 Enter 键。

过了一会儿，脚本返回了预期的数据。他检查了一下服务器，心想这也太容易了，结果所有该做的都完成了。“太酷了，真简单！”他说道，然后锁屏去吃午饭了。



# MySQL复制原理

一阵急促的敲门声把 Joel 吓了一跳, 又是老板来骚扰了。Joel 还没来得及说“请进”, 老板就已经走进来, 说道: “Joel, 有人抱怨说我们的响应时间变慢了, 想想怎么让它加速。管理员告诉我应用程序的读操作太多了, 看看能不能减少一些负载。”

Joel 还没来得及回答, Summerson 先生就出去了。“我想他的意思是我们需要一个更强大的服务器。” Joel 想。

好像知道 Joel 的想法一样, Summerson 先生又折回来进门说: “哦, 顺便说一句, 一开始我们买的所有服务器设备, 有一些还没用过, 你看看怎么处理。OK, Joel?” 说完他又走了。

“我在想我到底能不能搞定。” Joel 边想边将他心爱的 MySQL 书从书架上拿下来, 迅速浏览了目录, 找到“复制”这一章, 觉得可能有帮助吧。

如果正确使用 MySQL 的复制, 那么它就是一个非常有用的工具; 但如果复制出现故障, 或配置和使用不当, 就会相当令人头疼。本章从一个简单的配置开始, 涵盖 MySQL 复制的基础内容, 然后介绍一些基本技巧以丰富你的“复制工具箱”。

本章包括以下复制用例。

通过热备份达到高可用性

如果服务器宕机, 一切都将停止: 不能执行(可能很关键的)事务, 无法得到用户信息, 也不能检索其他重要数据。要不惜(几乎)一切代价避免这种情况发生, 因为它会严重破坏业务。最简单的方法就是配置一个额外的服务器专门作为热备份(hot standby), 在主服务器宕机的时候随时接管业务。

## 产生报表

直接用服务器上的数据创建报表将大大降低服务器的性能，在某些情况下尤其显著。如果产生报表需要大量的后台作业，最好创建一个额外的服务器来运行这些作业。停止报表数据库上的复制，然后在不影响主要业务服务器的情况下运行大量查询，从而得到数据库在某一特定时间的快照。例如，如果在每天最后一个事务处理完毕后停止复制，可以提取日报表而其他业务仍正常运转。

## 调试和审计

还可以审查服务器上的查询。例如，查看某些查询是否有性能问题，以及服务器是否由于某个糟糕的查询而不同步。

# 复制的基本步骤

本章将介绍一些最大化复制的效率和价值的尖端技术。首先我们要搭建一个如图 3-1 所示的简单复制，即单一的主从复制实例。这里不需要了解内部架构或复制过程的执行细节（在介绍更复杂的方案之前，我们先探讨这些内容）。

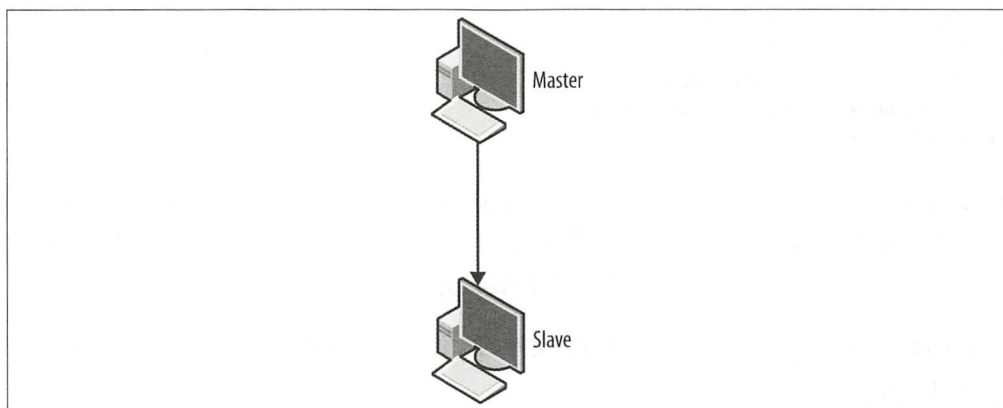


图3-1：简单的复制

建立基本的复制可以总结为以下三个简单步骤：

1. 配置一个服务器作为 master。
2. 配置一个服务器作为 slave。
3. 将 slave 连接到 master。

25

除非你从一开始就规划了复制且正确配置了 `my.cnf` 文件，否则步骤 1 和步骤 2 要求必须重启每个服务器。



执行这些步骤最简单的方法是拥有一个可以修改 *my.cnf* 文件权限的 shell 账号和一个具有 ALL 权限的服务器账号。<sup>注 1</sup>

必须严格限制在生产环境中授予权限。准确的指导方法请查阅本章后面的“配置复制的用户权限”。

## 配置 master

将服务器配置为 master，要确保该服务器有一个活动的二进制日志 (binary log) 和唯一的服务器 ID。我们后面再仔细研究二进制日志，现在只要知道二进制日志上保存了 master 上的所有变更记录，并且可以在 slave 上重新执行。服务器 ID 用于区分服务器。要创建二进制日志和服务器 ID，你需要停掉服务器，然后按示例 3-1 所示将 *log-bin*、*log-bin-index* 和 *server-id* 选项添加到 *my.cnf* 配置文件。加粗部分为添加的配置选项。

示例 3-1：向 *my.cnf* 添加选项以配置 master

```
[mysqld]
user      = mysql
pid-file  = /var/run/mysqld/mysqld.pid
socket    = /var/run/mysqld/mysqld.sock
port      = 3306
basedir   = /usr
datadir   = /var/lib/mysql
tmpdir    = /tmp
log-bin   = master-bin
log-bin-index = master-bin.index
server-id = 1
```

*log-bin* 选项给出了二进制日志产生的所有文件的基本名（稍后你将看到，二进制日志包含多个文件）。如果你创建了一个以 *log-bin* 为基本名的扩展文件名，该扩展名将被忽略，而只使用文件的基本名（也就是没有扩展的文件名）。

26 *log-bin-index* 选项给出了二进制索引文件的文件名，这个索引文件保存了所有 binlog 文件的列表。

严格地说，不需要为 *log-bin* 选项提供值，其默认值是 *hostname-bin*。*hostname* 的值来自 *pid-file* 选项，默认值是主机名（可以通过 *gethostname(2)* 系统调用得到）。如果管理员后来修改了主机名，binlog 文件名也会随之改变，但是索引文件仍可以获取正确的值。最好为 MySQL 服务器创建一个机器无关的唯一的服务器名，因 binlog 文件序列中途改名可能会很混乱。

如果没有为 *log-bin-index* 赋予任何值，其默认值与 binlog 文件的基本名相同（如果没有为 *log-bin* 提供值，则默认为 *hostname-bin*）。也就是说，如果你不赋值给 *log-bin*—

注 1 在 Windows 中，命令行工具 (CMD) 或 PowerShell 就相当于 UNIX 中的 shell。

index, 索引文件名会随主机名的改变而改变。所以, 如果你改变主机名然后重启服务器, 将找不到索引文件, 从而认为索引文件不存在, 导致二进制日志为空。

每个服务器都有一个唯一的服务器 ID, 所以如果一个 slave 连接了 master, 并且其 server-id 的参数值与 master 相同, 则会报 master 和 slave 服务器 ID 相同的错误。

设置完配置文件后重启服务器, 然后添加一个复制用户, 这样便完成了配置。

修改 master 的配置文件以后, 重启 master, 使配置生效。

slave 启动一个标准的客户端连到 master, 并请求 master 将所有的变更发送给它。slave 连接时要求 master 上有一个特殊复制权限的用户。示例 3-2 展示了 master 上的一个标准的 mysql 客户端会话, 通过命令创建新用户并赋予适当的权限。

示例3-2: 在master上创建一个复制用户

```
master> CREATE USER repl_user;
Query OK, 0 rows affected (0.00 sec)
master> GRANT REPLICATION SLAVE ON *.*
-> TO repl_user IDENTIFIED BY 'xyzyz';
Query OK, 0 rows affected (0.00 sec)
```



REPLICATION SLAVE 权限并没有什么特别之处, 不过拥有这个权限的用户能够获取 master 上的二进制日志。完全可以给一个常规用户赋予 REPLICATION SLAVE 权限, 但最好还是将复制 slave 用户与其他用户区别开来。这样的话, 以后如果想禁止某些 slave 的连接, 只要删除该用户就可以了。

27

## 配置 slave

配置完 master 后, 还需要配置 slave。与 master 一样, 需要为每个 slave 分配一个唯一的服务器 ID。分别使用 relay-log 和 relay-log-index 选项向 my.cnf 文件添加中继日志文件 (relay log file) 和中继日志索引文件 (relay log index file) 的文件名 (我们将在第 8 章的“复制架构基础”小节中详细讨论中继日志)。示例 3-3 给出了推荐的配置选项, 其中新添加的选项加粗表示。

示例3-3: 向my.cnf文件添加选项来配置slave

```
[mysqld]
user          = mysql
pid-file      = /var/run/mysqld/mysqld.pid
socket        = /var/run/mysqld/mysqld.sock
port         = 3306
```

```
basedir          = /usr
datadir          = /var/lib/mysql
tmpdir           = /tmp
server-id        = 2
relay-log-index  = slave-relay-bin.index
relay-log        = slave-relay-bin
```

与 log-bin 和 log-bin-index 选项一样，relay-log 和 relay-log-index 选项的默认值取决于 hostname。relay-log 的默认值是 *hostname-relay-bin*，relay-log-index 的默认值是 *hostname-relay-bin.index*。使用默认值有一个问题，即一旦服务器的主机名改变，将会因为无法找到中继日志索引文件而认为中继日志文件为空。

修改 *my.cnf* 文件以后，重启 slave，使配置生效。

## 配置复制的用户权限

要把 slave 连接到 master 做复制，除了需要一个能访问关键文件的 shell 账户，还要有一个具有特定权限的账户。出于安全原因，通常要严格控制这个账号只具备配置 master 和 slave 所需的必要权限。为了创建和删除用户，这个账号需要具备 CREATE USER 权限。为了将 REPLICATION SLAVE 权限赋予复制账号，还需要具有 GRANT OPTION 的 REPLICATION SLAVE 权限。

要进一步执行复制相关的工作（本章稍后将介绍），还需要更多选项：

- 执行 FLUSH LOGS 命令（或任何 FLUSH 命令）需要 RELOAD 权限。
- 执行 SHOW MASTER STATUS 和 SHOW SLAVE STATUS 命令需要 SUPER 或 REPLICATION CLIENT 权限。
- 执行 CHANGE MASTER TO 命令需要 SUPER 权限。

举个例子，向用户 mats 赋予足够的权限以完成本章中的所有工作，使用如下命令：

```
server> GRANT REPLICATION SLAVE, RELOAD, CREATE USER, SUPER
-> ON *.*
-> TO mats@'192.168.2.%'
-> WITH GRANT OPTION;
```

## 连接 master 和 slave

现在创建基本的复制只剩下最后一步了：将 slave 定向到 master，让它知道从哪里进行复制。为此，我们需要知道 master 的 4 部分信息：



- 主机名
- 端口号
- 具备 REPLICATION SLAVE 权限的用户账号
- 该用户账号的密码

在配置 master 的时候，我们已经创建了一个具备适当权限的用户账号及密码。主机名由操作系统确定，无法在 *my.cnf* 文件中配置，但端口号可以在 *my.cnf* 中分配（如果没有指定端口号，将使用默认值 3306）。创建和运行复制的最后两步是：使用 `CHANGE MASTER TO` 命令将 slave 定向到 master，然后用 `START SLAVE` 启动复制：

```
slave> CHANGE MASTER TO
->     MASTER_HOST = 'master-1',
->     MASTER_PORT = 3306,
->     MASTER_USER = 'repl_user',
->     MASTER_PASSWORD = 'xyzyzy';
Query OK, 0 rows affected (0.00 sec)
slave> START SLAVE;
Query OK, 0 rows affected (0.15 sec)
```

29

恭喜！你已经建立了第一个 master 与 slave 之间的复制。如果你在 master 的数据库上做一些改动，例如创建新表并填充数据，将发现这些变更都会复制到 slave。试试吧！建立一个测试数据库（如果还没有的话），再创建一些表，然后添加一些数据到表中，看看这些变更是否会复制到 slave 中。

注意，`MASTER_HOST` 参数的值可以是主机名或 IP 地址。如果是主机名，则可以通过调用 `gethostname(3)` 得到对应的 IP 地址，即通过域名查找来解析主机名，其结果与配置有关。配置域名解析的步骤不在本书讨论的范围内。

## 二进制日志简介

复制过程需要使用二进制日志（binary log，或者 binlog），它记录了服务器数据库上的所有变更。你需要理解二进制日志是如何控制复制过程或解决问题的，所以本节我们将介绍一些背景知识。

图 3-2 展示了复制架构示意图，图中包括 master、二进制日志和 slave，其中 slave 通过二进制日志获取 master 上的变更。我们将在第 8 章详细阐述复制架构。当某条语句即将执行结束时，将在二进制日志的末尾写入一条记录，同时通知语句解析器语句已经执行完毕。

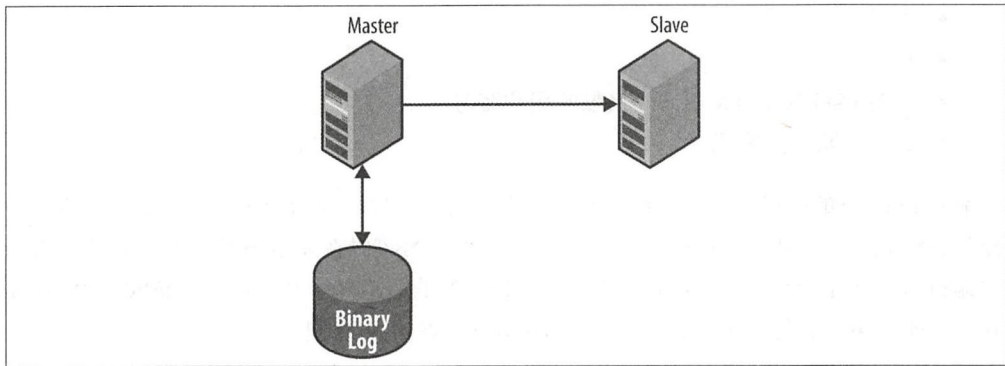


图3-2：二进制日志在复制中的作用

通常只有即将执行完毕的语句才会被写入二进制日志，但是在一些特殊情况下可能会写入其他信息，包括语句的附加信息或者直接代替语句被写入。不久你就会知道这么做的原因，但是目前我们假设只有执行的语句才会写入二进制日志。

## 二进制日志记录了什么

二进制日志的作用是记录数据库中表的更改，然后用于复制和 PITR（即时恢复，将在第 15 章中讨论），另外少数审计情况下也会用到。

注意，二进制日志只包括数据库的改动，所以对那些不改变数据的语句则不会写入二进制日志。

从传统意义上说，MySQL 复制记录了产生变化的 SQL 语句，称为基于语句的复制（statement-based replication）。由于基于语句的复制会在 slave 上重新执行语句，如果 master 和 slave 的上下文环境不完全一致的话，可能导致 slave 上的结果与 master 不同。所以在 5.1 版本中，MySQL 还提供了基于行的复制（row-based replication）。相比基于语句的复制，基于行的复制将每一条记录记为二进制日志中的一行。基于行的复制不仅更加方便，而且有时候速度更快。

考虑一个含有多表连接或 WHERE 条件的复杂更新，看看这两种复制有什么不同。你真正需要知道的是更新后数据的状态，而不是像基于语句的复制那样把所有的逻辑都在 slave 上重新执行。另一方面，如果某个更新改变了 10 000 行，你可能宁愿只记录这个语句，而不是像基于行的复制那样去记录 10 000 个单独的改动。

我们将在第 8 章涉及基于行的复制，介绍它的实现和使用。下面的例子重点讲解基于语句的复制，这样更容易理解数据库的执行。

## 观察复制的动作

用上一节中的复制例子，我们来看一些简单语句的 binlog 事件。先使用命令行客户端连接 master，然后执行一些命令获取二进制日志：

```
master> CREATE TABLE tbl (text TEXT);
Query OK, 0 rows affected (0.04 sec)
```

```
master> INSERT INTO tbl VALUES ("Yeah! Replication!");
Query OK, 1 row affected (0.00 sec)
```

```
master> SELECT * FROM tbl;
+-----+
| text          |
+-----+
| Yeah! Replication! |
+-----+
1 row in set (0.00 sec)
```

```
master> FLUSH LOGS;
Query OK, 0 rows affected (0.28 sec)
```

FLUSH LOGS 命令强制轮换 (rotate) 二进制日志，从而得到一个“完整的”二进制日志文件。使用 SHOW BINLOG EVENTS 命令进一步查看该文件，如示例 3-4 所示。

示例3-4：检查二进制日志中有哪些事件

```
master> SHOW BINLOG EVENTS\G
***** 1. row *****
  Log_name: mysql-bin.000001
    Pos: 4
Event_type: Format_desc
Server_id: 1
End_log_pos: 107
      Info: Server ver: 5.5.34-0ubuntu0.12.04.1-log, Binlog ver: 4
***** 2. row *****
  Log_name: mysql-bin.000001
    Pos: 107
Event_type: Query
Server_id: 1
End_log_pos: 198
      Info: use `test`; CREATE TABLE tbl (text TEXT)
***** 3. row *****
  Log_name: mysql-bin.000001
    Pos: 198
Event_type: Query
```

```

Server_id: 1
End_log_pos: 266
Info: BEGIN
***** 4. row *****
Log_name: mysql-bin.000001
Pos: 266
Event_type: Query
Server_id: 1
End_log_pos: 374
Info: use `test`; INSERT INTO tbl VALUES ("Yeah! Replication!")
***** 5. row *****
Log_name: mysql-bin.000001
Pos: 374
Event_type: Xid
Server_id: 1
End_log_pos: 401
Info: COMMIT /* xid=188 */
***** 6. row *****
Log_name: mysql-bin.000001
Pos: 401
Event_type: Rotate
Server_id: 1
End_log_pos: 444
Info: mysql-bin.000002;pos=4
6 rows in set (0.00 sec)

```

这个二进制日志包含 6 个事件：一个格式描述事件、三个查询事件，一个 XID 事件，以及一个日志轮换（rotate）事件。查询事件用于描述如何将数据库上执行的语句写入二进制日志，XID 事件用于事务管理，而格式描述事件和日志轮换事件则用于在服务器内部管理二进制日志。第 8 章将详细讨论这些事件，这里我们先简单看一下每个事件所包含的字段。

### Event\_type

这是事件的类型。这里我们已经看到了三种类型，但还有很多类型。事件的类型表明什么样的信息会被传送给 slave。目前在 MySQL 5.1.18 到 5.5.33 版本中，一共有 27 种事件类型（其中有些事件是不使用的，但是为了向后兼容而保留了），到 5.6.12 版本已经有 35 种事件类型了。这个范围是可扩展的，需要的话可以添加新的事件类型。

### Server\_id

这是创建事件的服务器的 ID。

### Log\_name

这是用来存储事件的文件名。一个事件只能存储在一个文件中，永远不能跨两个文件。

Pos

这是事件在文件中的开始位置，即事件的第一个字节。

End\_log\_pos

这是事件在文件中的结束位置，也是下一个事件的开始位置。这个位置比事件的最后一个字节高一位，因此事件的字节范围为 Pos 到 End\_log\_pos - 1。通过计算 End\_log\_pos - Pos 可以得到这个事件的长度。

Info

这是关于事件信息的可读文本。不同的事件会显示不同的信息，不过至少可以通过查询事件来知道它所包含的语句。

前两个字段 Log\_name 和 Pos 组成了事件的二进制日志位置 (binlog position)，用于标识事件的地点或位置。除了以上字段以外，每个事件还包括很多其他信息，例如时间戳，即从纪元（用经典的 UNIX 时间格式表示，例如 1970-01-01 00:00:00 UTC）开始的秒数。

## 二进制日志的结构和内容

33

前面讲过，二进制日志并不是一个单独的文件，而是由一系列易于管理的（例如在不影响新日志的情况下移除旧的日志）文件组成的。二进制日志包括一组存储实际内容的二进制日志文件和一个用来跟踪二进制日志文件存储位置的二进制日志索引文件。图 3-3 给出了二进制日志的组织结构。

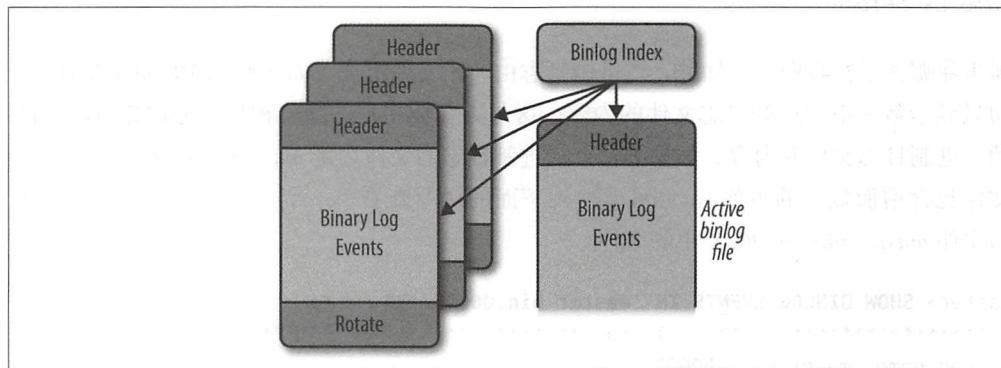


图3-3：二进制日志的结构

有一个二进制日志文件是活动二进制日志文件 (active binlog file)，即当前正在被写入的文件（通常也从这个文件读取）。

每个二进制日志文件都以格式描述事件 (format description event) 开始，以日志轮换事



件 (rotate event) 结束。格式描述日志事件包括产生该文件的服务器版本号、服务器及二进制日志的信息等。日志轮换事件包含下一个二进制日志文件的名称,以告知二进制日志继续写入哪个文件。

每个二进制日志文件中有多个二进制日志事件,各个事件之间相互独立,同时也是构成二进制日志的基本单位。格式描述日志事件还有一个标记,标记二进制日志文件是否正常关闭。如果正在写入二进制日志文件,则设置该标记;如果文件关闭,则清除该标记。这样就可以检测出在崩溃事件中损坏的二进制日志文件,并允许通过复制进行恢复。

如果在 master 上执行其他语句,你会发现一件奇怪的事情——看不到二进制日志中的变化:

```
master> INSERT INTO tbl VALUES ("What's up?");
Query OK, 1 row affected (0.00 sec)
```

```
master> SELECT * FROM tbl;
```

```
+-----+
| text          |
+-----+
```

```
| Yeah! Replication! |
+-----+
```

```
| Yeah! Replication! |
+-----+
```

```
| What's up?        |
+-----+
```

```
| What's up?        |
+-----+
```

```
1 row in set (0.00 sec)
```

```
master> SHOW BINLOG EVENTS\G
```

```
same as before
```

新事件哪去了? 我们已经知道,二进制日志由若干文件组成,而 SHOW BINLOG EVENTS 语句只显示第一个二进制日志文件的内容。这与大多数用户的期望相反,他们想看的是活动二进制日志文件的内容。如果第一个二进制日志的文件名是 *master-bin.000001* (这个文件包含前面显示的事件),你可以使用下面的命令查看下一个二进制日志文件 (这里即文件 *master-bin.000002*) 中的事件:

```
master> SHOW BINLOG EVENTS IN 'master-bin.000002'\G
```

```
***** 1. row *****
```

```
Log_name: mysql-bin.000002
```

```
Pos: 4
```

```
Event_type: Format_desc
```

```
Server_id: 1
```

```
End_log_pos: 107
```

```
Info: Server ver: 5.5.34-0ubuntu0.12.04.1-log, Binlog ver: 4
```

```

***** 2. row *****
Log_name: mysql-bin.000002
Pos: 107
Event_type: Query
Server_id: 1
End_log_pos: 175
Info: BEGIN
***** 3. row *****
Log_name: mysql-bin.000002
Pos: 175
Event_type: Query
Server_id: 1
End_log_pos: 275
Info: use `test`; INSERT INTO tbl VALUES ("What's up?")
***** 4. row *****
Log_name: mysql-bin.000002
Pos: 275
Event_type: Xid
Server_id: 1
End_log_pos: 302
Info: COMMIT /* xid=196 */
4 rows in set (0.00 sec)

```

你可能已经注意到在示例 3-4 中，二进制日志以日志轮换事件结尾，Info 字段包含下一个二进制日志文件名和事件的开始位置。使用 SHOW MASTER STATUS 命令查看当前正在写入的是哪个二进制日志文件：

```

master> SHOW MASTER STATUS\G
***** 1. row *****
File: master-bin.000002
Position: 205
Binlog_Do_DB:
Binlog_Ignore_DB:
1 row in set (0.00 sec)

```

35

查看二进制日志以后，停止并重置 slave，然后删除表：

```

master> DROP TABLE tbl;
Query OK, 0 rows affected (0.00 sec)

slave> STOP SLAVE;
Query OK, 0 rows affected (0.08 sec)

slave> RESET SLAVE;
Query OK, 0 rows affected (0.00 sec)

```

接着，删除表，然后重置 master 刷新：

```
master> DROP TABLE tbl;  
Query OK, 0 rows affected (0.00 sec)
```

```
master> RESET MASTER;  
Query OK, 0 rows affected (0.04 sec)
```

RESET MASTER 命令删除了所有二进制日志文件并清空了二进制日志索引文件。RESET SLAVE 命令删除了 slave 上复制用的所有文件，重新开始。



无论是 RESET MASTER 还是 RESET SLAVE 都要求没有活动的复制正在运行。因此：

- 执行 RESET MASTER 命令（在 master 上）时，确保没有 slave 连接到该 master。
- 执行 RESET SLAVE 命令（在 slave 上）时，先执行 STOP SLAVE 命令，确保 slave 上没有活动的复制。

本章涵盖了大多数基本事件，而完整详细的事件列表请参考 MySQL 内部手册 (<http://bit.ly/mysql-manual>)。

## 建立新 slave

既然你已经对二进制日志有所了解，下面我们要解决之前创建 slave 的过程中存在的一个基本问题。前面配置 slave 时，我们并没有说明复制从哪里开始，所以 slave 将从头开始读取 master 上的二进制日志。如果 master 已经运行了一段时间，这显然不是个好方法：不仅会在 slave 上大量重放事件，还可能导致有些日志无法获取，因为它们可能由于安全原因存储在其他地方，master 上也没有这些日志信息（第 15 章介绍备份和 PITR 时将进一步讨论这个）。

所以我们需要另一种方法来建立新的 slave（又称自举 slave），而不是从头开始复制。

这里 CHANGE MASTER TO 命令有两个有用的参数，即：MASTER\_LOG\_FILE 和 MASTER\_LOG\_POS。（从 MySQL 5.6 开始，又出现了另一种更简便的方式来指定这些位置，包括全局事务标识符，GTID 等，更多内容参见第 8 章。）使用这些参数指定 master 开始发送事件的 binlog 位置，而不是从头开始。

向 CHANGE MASTER TO 命令添加上述参数，按照以下步骤建立新的 slave：

1. 配置新的 slave。

2. 备份 master (或者备份已经复制了 master 的 slave)。参见第 15 章中的常用备份技术。
3. 记下该备份相应的 binlog 位置(即产生 master 当前状态的最后一个事件所在的位置)。
4. 在新 slave 上恢复备份。参见第 15 章中的常用恢复技术。
5. 配置 slave 从这个 binlog 位置开始复制。

根据第 2 步使用的是 master 还是 slave, 处理过程略有差异。我们先看只有一个服务器且作为 master 运行时, 如何引导新 slave, 这个过程称为克隆 master。

克隆 master 是指获取服务器的快照, 通常通过创建备份来完成。服务器的备份技术有很多, 但本章仅使用较为简单的技术, 即运行 *mysqldump* 来创建逻辑备份。其他备份技术包括: 通过复制数据库文件创建物理备份, 诸如 MySQL 企业备份工具的在线备份技术, 以及使用 Linux 的 LVM (Logical Volume Manager, 逻辑卷管理器) 进行卷快照等。这些技术将在第 15 章进行详细介绍, 并讨论它们各自的优点。

## 克隆 master

虽然 *mysqldump* 工具可以一步完成所有步骤, 但为了解释必要的操作, 这里我们将单独执行每个步骤。本节后面会提供一个更紧凑的版本。

如图 3-4 所示, 克隆 master 首先要创建 master 的备份。由于 master 可能正在运行, 而且缓存中有很多表, 所以需要刷新 (flush) 所有表并锁定数据库, 防止在检查 binlog 位置之前数据库发生改变。使用 `FLUSH TABLES WITH READ LOCK` 命令来完成:

```
master> FLUSH TABLES WITH READ LOCK;  
Query OK, 0 rows affected (0.02 sec)
```

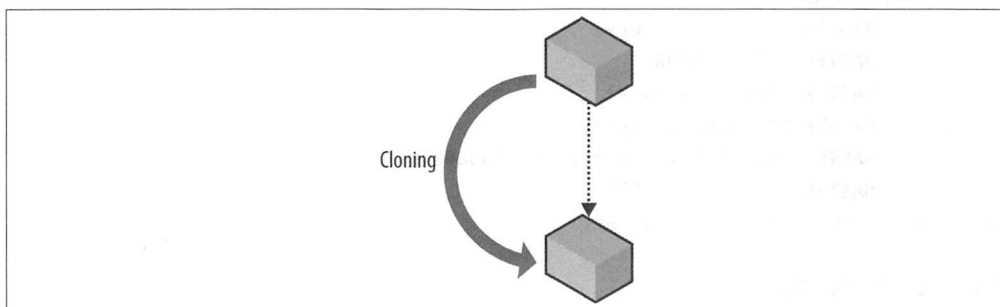


图3-4: 克隆master来创建新的slave

一旦数据库被锁定, 就可以创建备份, 并记录 binlog 位置了。注意, 这时不应该断开服

务器连接，因为那会导致锁被释放。由于 master 没有任何改动，SHOW MASTER STATUS 命令将正确返回当前二进制日志文件及其 binlog 位置。我们将在第 8 章详细讨论 SHOW MASTER STATUS 和 SHOW MASTER LOGS 命令。

```
master> SHOW MASTER STATUS\G
***** 1. row *****
      File: master-bin.000042
      Position: 456552
      Binlog_Do_DB:
      Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

下一个事件的写入位置是 master-bin.000042:456552，这就是复制的起点，这个位置点之前的所有东西都在备份里。记下 binlog 位置后就可以创建备份了。创建数据库备份最简单的方法是用 *mysqldump*：

```
$ mysqldump --all-databases --host=master-1 >backup.sql
```

38 有了可靠的 master 副本，就可以为数据库中的表解锁，允许数据库继续处理查询。

```
master> UNLOCK TABLES;
Query OK, 0 rows affected (0.23 sec)
```

接下来，在 slave 上使用 *mysql* 实用工具恢复备份：

```
$ mysql --host=slave-1 <backup.sql
```

已经在 slave 上恢复了 master 的备份，现在可以启动 slave 了。利用前面记下的 master 的 binlog 位置，使用 CHANGE MASTER TO 命令配置 slave，然后启动 slave：

```
slave> CHANGE MASTER TO
->     MASTER_HOST = 'master-1',
->     MASTER_PORT = 3306,
->     MASTER_USER = 'slave-1',
->     MASTER_PASSWORD = 'xyzyzy',
->     MASTER_LOG_FILE = 'master-bin.000042',
->     MASTER_LOG_POS = 456552;
Query OK, 0 rows affected (0.00 sec)
```

```
slave> START SLAVE;
Query OK, 0 rows affected (0.25 sec)
```

*mysqldump* 可以自动执行前面的所有步骤。要对 master 服务器上的所有数据库进行逻辑备份，输入：



```
$ mysqldump --host=master --all-databases \  
> --master-data=1 >backup-source.sql
```

--master-data=1 选项使 *mysqldump* 产生 CHANGE MASTER TO 语句，其参数为二进制日志文件及其位置（可通过 SHOW MASTER STATUS 得到）。

然后就可以在 slave 上恢复备份了：

```
$ mysql --host=slave-1 <backup-source.sql
```

注意，只有克隆 master 的时候能用 --master-data=1 自动执行 CHANGE MASTER TO 语句。后面克隆 slave 的时候，这些步骤都要分开执行。

恭喜！现在你已经克隆了 master，并建立和运行了一个新 slave。根据 master 的负载情况，你可能需要 slave 从前面记录的位置开始同步，这比从头开始容易得多。

根据备份需要的时间长短不同，可能有大量的数据需要同步，所以，在将 slave 联机（online）之前，首先要仔细阅读第 6 章的“数据的一致性管理”一节。

## 克隆 slave

39

只要有一个 slave 连在 master 上，就可以使用这个 slave 创建新的 slave，而不需要再离线（offline）master 了。如果数据库很大或访问量较高，那么停机时间可能会相当长，因为既要考虑创建备份的时间又要考虑 slave 同步的时间。

克隆 slave 的过程如图 3-5 所示，与克隆 master 基本相同，区别在于如何找到 binlog 位置。另外，需要注意你克隆的那个 slave 同时还在执行从 master 的复制。

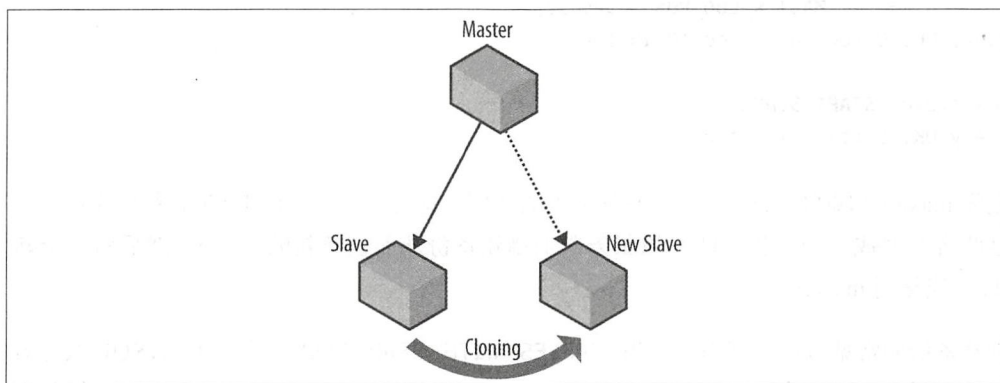


图3-5：克隆slave以创建一个新的slave

首先，必须在备份前停止 slave，保证 slave 上不再有变化发生。如果创建备份时复制仍

在进行，而在备份期间数据库又发生了变化，这时就会得到不一致的备份映像。但是，如果使用某种在线备份方法，如 MySQL 企业备份工具，则不需要在创建备份前停止 slave。停止 slave 是这样的：

```
original-slave> STOP SLAVE;  
Query OK, 0 rows affected (0.20 sec)
```

slave 停止以后，就可以像从前一样刷新数据表，然后创建备份。创建了 slave 的备份（而不是 master 的备份）后，使用 SHOW SLAVE STATUS 命令（而不是 SHOW MASTER STATUS）来确定从哪里开始复制。该命令会输出很多内容，第 8 章将详细介绍。如果要获得 master 二进制日志中 slave 即将执行的下一个事件的位置，请注意 Relay\_Master\_Log\_File 和 Exec\_Master\_Log\_Pos 字段的值。

```
40 original-slave> SHOW SLAVE STATUS\G  
...  
Relay_Master_Log_File: master-bin.000042  
...  
Exec_Master_Log_Pos: 546632
```

创建备份，然后在新 slave 上恢复备份以后，将复制的起点配置为从这个位置开始，然后启动新的 slave：

```
new-slave> CHANGE MASTER TO  
-> MASTER_HOST = 'master-1',  
-> MASTER_PORT = 3306,  
-> MASTER_USER = 'slave-1',  
-> MASTER_PASSWORD = 'xyzyz',  
-> MASTER_LOG_FILE = 'master-bin.000042',  
-> MASTER_LOG_POS = 546632;  
Query OK, 0 rows affected (0.19 sec)
```

```
new-slave> START SLAVE;  
Query OK, 0 rows affected (0.24 sec)
```

克隆 master 和克隆 slave 只有一些细节上的区别，这意味着我们的 Python 程序库可以将这两者合并成一个过程，即在源服务器上创建备份从而创建新的 slave，然后将这个新 slave 连接到 master。

创建备份的常见方法是调用 FLUSH TABLES WITH READ LOCK，然后在 MySQL 服务器被读锁锁住的时候创建一份数据库文件的副本。这通常比 *mysqldump* 快很多，但是在 InnoDB 中使用 FLUSH TABLES WITH READ LOCK 是不安全的！

FLUSH TABLES WITH READ LOCK 会锁住表，阻止新事务发生，但后台仍然有一些 FLUSH TABLES WITH READ LOCK 阻止不了的活动在继续进行。

使用下面的方法可安全地创建 InnoDB 数据表的备份：

- 关闭服务器，然后复制文件。如果数据库很大，最好采取这种方法，因为这时使用 *mysqldump* 进行数据恢复会很慢。
- 执行 FLUSH TABLES WITH READ LOCK（前面已经介绍过）命令后，使用 *mysqldump* 工具。读锁保证读取数据的时候不会产生变更。如果有大量数据要读的话，数据库可能长时间处于被锁的状态。注意，使用 `--single-transaction` 选项也可以获得一致的快照，但只对 InnoDB 表有效。更多内容参见第 15 章的“*mysqldump* 工具”一节。
- 执行 FLUSH TABLES WITH READ LOCK 锁定数据库后，执行某种快照方案，例如 LVM（Linux 平台）或者 ZFS（Zettabyte File System）。
- 使用 MySQL 企业备份工具（或 XtraBackup）做 MySQL 联机备份。

41

## 克隆操作的脚本

Python 程序库实现克隆 master 的方法很简单，使用一个 Server 对象代表 master，然后从这个 master 上复制数据库即可。使用 clone 函数即可实现，见示例 3-6。

克隆 slave 的过程类似，但它是在一台服务器上创建备份，然后新 slave 连接到另一台服务器进行复制。同时支持 master 和 slave 的克隆操作很容易，只需要使用两个参数：source 参数指定从哪里创建备份，use\_master 参数指出备份恢复后 slave 需要连接的服务器。clone 方法的调用格式是这样的：

```
clone(slave = slave[1], source = slave[0], use_master = master)
```

下面要写一些实用工具函数来实现克隆功能，这些函数在其他地方也可以派上用场。示例 3-5 用到了如下函数：

fetch\_master\_pos

从 master 获取 binlog 位置（即 master 即将写入二进制日志的下一个事件的位置）。

fetch\_slave\_pos

从 slave 获取 binlog 位置（即从 master 读取的下一个事件的位置）。

replicate\_from

需要提供的参数包括 slave、master 及 binlog 位置，将 slave 定向到 master，并从给定的 binlog 位置开始复制。

`replicate_from` 函数从 `master` 上读取 `repl_user` 字段获取复制用户的用户名和密码。但 `Server` 类的定义中并没有该字段，它是服务器注入 `Master` 角色时添加的字段。

示例3-5：获取服务器的`master`和`slave`位置的功能函数

```
_CHANGE_MASTER_TO = """CHANGE MASTER TO
    MASTER_HOST=%s, MASTER_PORT=%s,
    MASTER_USER=%s, MASTER_PASSWORD=%s,
    MASTER_LOG_FILE=%s, MASTER_LOG_POS=%s"""

def replicate_from(slave, master, position):
    slave.sql(_CHANGE_MASTER_TO, (master.host, master.port,
                                   master.repl_user.name,
                                   master.repl_user.passwd,
                                   position.file, position.pos))

42 def fetch_master_pos(server):
    result = server.sql("SHOW MASTER STATUS")
    return Position(server.server_id, result["File"], result["Position"])

def fetch_slave_pos(server):
    result = server.sql("SHOW SLAVE STATUS")
    return Position(server.server_id, result["Relay_Master_Log_File"],
                    result["Exec_Master_Log_Pos"])
```

这些都是创建 `clone` 函数所需要的函数。如果要克隆 `slave`，应用程序传递一个单独的 `use_master` 参数，让 `clone` 函数将新 `slave` 定向到该 `master` 做复制。如果要克隆 `master`，应用程序则忽略这个 `use_master` 参数，让 `clone` 函数使用“`source`”服务器作为 `master`。

创建服务器备份的方法有很多，示例 3-6 仅选择了一种方法，即使用 `mysqldump` 创建服务器的逻辑备份。稍后我们将演示如何将这个备份过程通用化，使得无论采用任何备份方法都可以使用相同的基本代码建立新 `slave`。

示例3-6：克隆`master`或`Slave`的函数

```
def clone(slave, source, use_master = None):
    from subprocess import call
    backup_file = open(server.host + "-backup.sql", "w+")
    if master is not None:
        source.sql("STOP SLAVE")
    lock_database(source)
    if master is None:
        position = fetch_master_position(source)
    else:
        position = fetch_slave_position(source)
    call(["mysqldump", "--all-databases", "--host='%s'" % source.host],
```



```

        stdout=backup_file)
if master is not None:
    start_slave(source)
backup_file.seek() # Rewind to beginning
call(["mysql", "--host='%s'" % slave.host], stdin=backup_file)
if master is None:
    replicate_from(slave, source, position)
else:
    replicate_from(slave, master, position)
start_slave(slave)

```

## 执行常见的复制任务

每个常见的复制案例，包括横向扩展（scale out）、热备份（hot standby）等，都有其实现细节和可能的陷阱。下面将向读者展示如何执行某些任务，以及如何使 Python 程序库支持这些任务。



本节中的例子省略了密码。配置某些控制服务器的账户时，可以设置只允许某些控制部署的特定主机访问（通过创建类似 `mats@'192.168.2.136'` 这样的账户），或者也可以给这些命令设置密码。

43

## 报表

大多数企业需要大量的例行报告，包括已售物品的周报表，开支和收入的月报表，以及大量的数据挖掘报表，以发现某些趋势或为营销部门识别重点群体等。

在 master 上运行这些查询会很麻烦。数据挖掘查询可能需要大量的计算资源，拖慢正常操作，而结果却发现不值得那么做，比如把左撇子作为重点群体。此外，这些报表通常不是很紧急（与处理日常事务相比），所以没有必要尽快创建报表。换句话说，这些报表对时间要求不严格，即使一个小时不够，花两个小时完成也没什么关系。

更好的办法是，使用一个空闲的服务器（或者两个，如果真有那么多样报表需求的话），搭建一个到 master 的复制。如果需要做报表的话，就停止复制，执行报表程序，然后再重启复制，这样完全不会干扰 master。

报表往往需要精确的间隔时间，如汇总当天所有的销售额，需要在适当的时候停止复制，而不至于将第二天的销售额也计入前一天的报表。然后，没有特定日期或时间的事件发生，就无法停止 slave，所以必须得想其他的方法。



假设每天做一次报表，包括所有从午夜 12 点到下一个午夜 12 点的交易。需要在午夜 12 点停止报表 slave，确保 12 点以后 slave 上不再有新的事件执行而 12 点以前的所有事件都已经执行完毕。我们并不想手动执行此操作，所以考虑如何使这个过程自动化。可按照以下步骤执行：

1. 12 点前，也可能是 12 点前 5 分钟，停止报表 slave，确保不会再从 master 接收事件。
2. 12 点后，检查 master 上的二进制日志，找到 12 点前记录的最后一个事件。显然，如果在 12 点之前进行这一步操作，可能无法得到当天的所有事件。
3. 记录该事件的 binlog 位置，然后启动 slave。
4. 直到 slave 同步到这个位置后停止。

第一个问题是如何正确地调度任务。调度任务的方法很多，与操作系统有关。这里我们不会详细讨论这个问题，但你可以在本章后面的“在 UNIX 上任务调度”一节中看到如何在类 UNIX 操作系统（如 Linux）中调度任务。

停止 slave 很简单，执行 STOP SLAVE 命令，然后在 slave 停止后记录 binlog 位置：

```
slave> STOP SLAVE;
Query OK, 0 rows affected (0.25 sec)

slave> SHOW SLAVE STATUS\G
...
Relay_Master_Log_File: capulet-bin.000004
...
Exec_Master_Log_Pos: 2456
1 row in set (0.00 sec)
```

其他三个步骤要在实际报表开始前执行，通常作为实际报表脚本的一部分。在描述脚本之前，我们先考虑每个步骤是怎样执行的。

调用 `mysqlbinlog` 命令读取二进制日志的内容。这一点后面会详细介绍，第二步中使用了这个工具。`mysqlbinlog` 命令提供了两个选项，用于部分读取二进制日志，即 `--start-datetime` 和 `--stop-datetime`。因此，要获得从停止 slave 时刻到 12 点期间的所有事件，可以使用以下命令：

```
$ mysqlbinlog --force --read-from-remote-server --host=reporting.bigcorp.com \
> --start-datetime='2009-09-25 23:55:00'
> --stop-datetime='2009-09-25 23:59:59' \
> binlog files
```

事件中存储的时间戳是指语句开始执行的时间戳，而不是它写入二进制日志的时间戳。

--stop-datetime 选项说明在指定 date/time 后的第一个时间戳时刻停止产生事件。可能有些事件，它们在 date/time 之前开始执行，但却在 date/time 后才写入二进制日志，这样的事件不包括在内。

由于这时 master 正在写二进制日志，需要提供 --force 选项。否则，mysqlbinlog 无法读取已打开的二进制日志。这个命令需要提供一组 binlog 文件作为参数。由于这些文件的名称取决于配置选项，所以只能从服务器上获取这些文件名。然后，要搞清楚 mysqlbinlog 命令需要哪些 binlog 文件。使用 SHOW BINARY LOGS 命令可以很容易地获取 binlog 文件名列表：

```
master> SHOW BINARY LOGS;
```

```
+-----+-----+
| Log_name|File_size |
+-----+-----+
| capulet-bin.000001 |      24316 |
| capulet-bin.000002 |       1565 |
| capulet-bin.000003 |        125 |
| capulet-bin.000004 |       2749 |
+-----+-----+
4 rows in set (0.00 sec)
```

本例只有 4 个 binlog 文件，但其实 binlog 文件可以有很多。在停止 slave 前扫描这么多文件只是浪费时间，所以最好能减少需要读取的文件数量，以便确定 slave 停止的正确位置。由于第 1 步中已经记下了 binlog 位置，由此可知 slave 停止运行时的 binlog 文件名，然后将该文件名及其后面所有的文件名作为 mysqlbinlog 实用工具的输入。通常输入参数只有一个文件（或者有两个，比如在关闭 slave 之后、启动报表事件之前的二进制日志轮换时）。

当使用 mysqlbinlog 命令解析少数几个 binlog 文件时，将输出事件相关的文本信息。

```
$ mysqlbinlog --force --read-from-remote-server --host=reporting.bigcorp.com \
> --start-datetime='2009-09-25 23:55:00'
> --stop-datetime='2009-09-25 23:59:59' \
> capulet-bin.000004
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 4
#090909 22:16:25 server id 1 end_log_pos 106 Start: binlog v 4, server v...
ROLLBACK/*!*/;
.
.
.
```

```
# at 2495
#090929 23:58:36 server id 1 end_log_pos 2650 Query thread_id=27 exe...
SET TIMESTAMP=1254213690/*!*/;
SET /*!*/;
INSERT INTO message_board(user, message)
VALUES ('mats@sun.com', 'Midnight, and I'm bored')
/*!*/;
```

有趣的是，最后一个事件的 `end_log_pos` 参数（本例中该值为 2650），就是 12 点后的下一个事件即将被写入的位置。

如果注意观察前面的命令输出结果，你会发现我们并不知道这个字节位置指向哪个 binlog 文件，而 `mysqlbinlog` 需要指定一个文件来查找这个事件。如果 `mysqlbinlog` 命令的参数是单个文件，显然就用这个文件查找；但如果参数是两个文件，则必须确定当天最后一个事件在哪个文件里。

观察含有 `end_log_pos` 的那行，还会看到事件类型信息。每个 binlog 文件都是以格式描述事件开始，这些事件会出现在前面的命令输出结果中。检查这些事件，就可以确定要查找的事件的位置。如果有两个格式描述事件，则该事件在第二个文件中；如果只有一个格式描述事件，则在第一个文件中。

报表工作开始前的最后一步是启动复制，然后在正确的位置停止它，即午夜 12 点以后的事件写入位置（即将写入或已被写入）。可以使用不太常用的命令 `START SLAVE UNTIL` 来完成。该命令的参数包括 master 日志文件，以及 slave 停止时对应的 master 日志位置。一旦 slave 到达指定位置，就会自动停止。

```
report> START SLAVE UNTIL
-> MASTER_LOG_POS='capulet-bin.000004',
-> MASTER_LOG_POS=2650;
Query OK, 0 rows affected (0.18 sec)
```

与 `STOP SLAVE` 命令（不含 `UNTIL`）一样，`START SLAVE UNTIL` 命令也会立即返回，但是如果 slave 已到达停止位置则不再返回。所以，只要 slave 还在运行，在 `STOP SLAVE UNTIL` 之后发出的命令仍会继续执行。使用 `MASTER_POS_WAIT` 函数等待 slave 到达停止位置，该函数会阻塞执行直到 slave 到达指定位置。

```
report> SELECT MASTER_POS_WAIT('capulet-bin.000004', 2650);
Query OK, 0 rows affected (231.15 sec)
```

现在 slave 在当天最后一个事件处停止了，报表过程可以开始分析数据和生成报告了。

## 使用 Python 处理报表

使用 Python 进行自动化报表处理是很简单的。示例 3-7 给出了在适当的时候停止报表的代码。

`fetch_remote_binlog` 函数通过 `mysqlbinlog` 命令从远程服务器读取二进制日志。将文件内容作为迭代器返回，遍历文件中的各行。还可以提供一个文件列表以优化读取。还可以指定开始的日期 / 时间和结束的日期 / 时间，以限制返回结果的日期 / 时间范围。这些都会作为参数传递给 `mysqlbinlog` 程序。

`find_datetime_position` 函数逐行扫描 binlog 文件直到找到 `end_log_pos`，同时记录观察到的开始事件个数。该函数还联系报表服务器，以确定何时停止读取 binlog 文件，然后联系 master 获取 binlog 文件，并确定从哪个 binlog 文件开始扫描。

◀ 47

示例3-7: 在指定时间前运行复制的Python代码

```
def fetch_remote_binlog(server, binlog_files=None,
                        start_datetime=None, stop_datetime=None):
    from subprocess import Popen, PIPE
    if not binlog_files:
        binlog_files = [
            row["Log_name"] for row in server.sql("SHOW BINARY LOGS")]

    command = ["mysqlbinlog",
               "--read-from-remote-server",
               "--force",
               "--host=%s" % (server.host),
               "--user=%s" % (server.sql_user.name)]

    if server.sql_user.passwd:
        command.append("--password=%s" % (server.sql_user.passwd))
    if start_datetime:
        command.append("--start-datetime=%s" % (start_datetime))
    if stop_datetime:
        command.append("--stop-datetime=%s" % (stop_datetime))
    return iter(Popen(command + binlog_files, stdout=PIPE).stdout)


def find_datetime_position(master, report, start_datetime, stop_datetime):
    from itertools import dropwhile
    from mysql.replicant import Position
    import re

    all_files = [row["Log_name"] for row in master.sql("SHOW BINARY LOGS")]
    stop_file = report.sql("SHOW SLAVE STATUS")["Relay_Master_Log_File"]
    files = list(dropwhile(lambda file: file != stop_file, all_files))
    lines = fetch_remote_binlog(server, binlog_files=files,
```



```

                                start_datetime=start_datetime,
                                stop_datetime=stop_datetime)

binlog_files = 0
last_epos = None
for line in lines:
    m = re.match(r"#\d{6}\s+\d?\d:\d\d:\d\d\s+"
                  r"server id\s+(?P<sid>\d+)\s+"
                  r"end_log_pos\s+(?P<epos>\d+)\s+"
                  r"(?P<type>\w+)", line)

    if m:
        if m.group("type") == "Start":
            binlog_files += 1
        if m.group("type") == "Query":
            last_epos = m.group("epos")
    return Position(files[binlog_files-1], last_epos)

```

现在你可以使用这些函数在真正执行报表任务之前同步报表服务器。

```

48 master.connect()
   report.connect()
   pos = find_datetime_position(master, report,
                                start_datetime="2009-09-14 23:55:00",
                                stop_datetime="2009-09-14 23:59:59")
   report.sql("START SLAVE UNTIL MASTER_LOG_FILE=%s, MASTER_LOG_POS=%s",
              (pos.file, pos.pos))
   report.sql("DO MASTER_POS_WAIT(%s,%s)", (pos.file, pos.pos))
   .
   .
code for reporting
   .
   .

```

可见，复制工作是非常简单的。这个例子涉及了一些关键概念，我们将在后面讨论横向扩展的时候用到它们，包括如何在正确的时间启动和停止 slave，如何获取 binlog 位置信息或怎样使用标准工具实现，以及如何整合得出自动化解决方案以满足特定需要。

## 在 UNIX 上调度任务

为了确保在午夜 12 点前停止 slave，然后在 12 点后启动报表，最简单的方法就是建立一个 *cron(8)* 作业，向 slave 发送停止运行的命令然后启动报表脚本。

例如，下面的 *crontab(5)* 作业可以保证 slave 在午夜 12 点前停止，12 点后运行报表脚本让 slave 前滚（roll forward），比如说，12 点 05 分。这里我们假设 *stop\_slave* 脚本用于停止 slave，*daily\_report* 脚本用于运行日报表（首先要同步报表服务器，如前所述）。



```
# stop reporting slave five minutes before midnight, every day
55 23 * * * $HOME/mysql_control/stop_slave

# Run reporting script five minutes after midnight, every day
5 0 * * * $HOME/mysql_control/daily_report
```

这些脚本默认放在一个名为 *reporttab* 的 crontab 文件中，使用 `crontab reporttab` 命令可以安装这个 crontab 文件。

## 在 Windows 上调度任务

打开“运行”（Windows 键 + R）对话框，输入 *taskschd.msc*，打开任务计划程序（Task Scheduler）。根据安全设置及 Windows 版本不同，可能会弹出用户账户控制（UAC）对话框，单击“继续”按钮。在“操作”菜单中选择“创建基本任务”，将会创建一个由时间触发的任务。该指令会打开“创建基本任务向导”，其中包含一些创建简单任务的步骤。第一步是为任务命名，并提供一个可选的描述。然后单击“下一步”按钮。

49

第二个页面指定任务的频率。有很多控制任务何时运行的选项，包括：一次、每天、每周、当前用户登录时，或者当指定事件被记录时。选择完后单击“下一步”按钮。根据选择的频率不同，第三个页面继续指定相应的触发细节（例如日期和时间）。设置完触发器后单击“下一步”按钮。

第四个页面设置任务被触发时执行什么操作，包括“启动程序”、“发送电子邮件”或者“显示消息”。选择其中一项，单击“下一步”按钮进入下一个页面。根据刚才的选择出现相应的页面，以设置具体的操作。例如，如果选择“启动程序”项，你可能要输入程序或脚本的文件名，添加参数及该任务从哪里开始等。

一旦所有信息填写完毕，单击“下一步”按钮，最后一个页面是对任务的重新检查。如果一切都没问题，单击“完成”按钮，任务就被创建了。如果需要返回前面的页面更改设置，则单击“上一步”按钮。如果选中“当单击完成时，打开此任务属性的对话框”项，可更改任务属性。

## 小结

本章介绍了 MySQL 复制，包括为什么要使用复制，以及如何建立复制。我们还简单介绍了二进制日志。下一章将进一步研究二进制日志。

Joel 刚刚把如何对 4 个新添加的 slave 进行负载均衡的报告，以及怎样扩展拓扑结构以应对未来需求的计划交给了 Summerson 先生。

“做得好，Joel。再解释一下，什么是 slave 来着。”

Joel 想叹气却又忍住了，说：“slave 是数据库服务器上的数据副本，它从一个叫 master 的原始服务器上获取变更，然后进行数据复制……”

## 二进制日志

“Joel ? ”

Joel 吓了一跳，赶紧从桌子底下爬出来，差点撞到头。他解释说：“我刚刚在整理一些线。”

Summerson 先生稍微点点头，然后很郑重地说：“市场部的新服务器有点问题，需要你解决一下。他们要把数据回滚到某个特定的时间点。”

“嗯，那要看具体情况了……” Joel 刚要说点什么，他担心是否有系统以前状态的快照。

“我告诉他们你能搞定。”

说完 Summerson 先生就转身走开了。过了一会儿，一个女人站在他门口，说：“他总是这样，并不是针对你。我们都把这叫作过路任务。”她笑了，然后做了自我介绍：“我叫 Amy，在这做开发的。”

Joel 绕过办公桌走到门口，说“我叫 Joel。”

片刻尴尬的沉默后，Joel 说：“呃，我应该开始那个任务了。”

Amy 笑着说：“一会儿见。”

“只要专注于达成目标需要做的事就好。” Joel 想。他走回办公桌前，找他上周买的那本 MySQL 的书。

上一章简单介绍了二进制日志。本章我们将深入细节，详细描述二进制日志的结构，复制事件的格式，以及如何使用 `mysqlbinlog` 工具来审查和处理二进制日志的内容。

二进制日志记录了数据库的所有变化。通常二进制日志用于复制，所以会在 slave 上执行同样的更新。由于二进制日志保留了所有更新记录，可以用于审计目的，看看在数据库中发生了什么；还可用于 PITR（基于时间点恢复），即向服务器回放二进制日志，重新执行二进制日志中记录的更新。（这其实就是第 3 章“报表”那一节做的事情，我们将 23:55:00 到 23:59:59 之间所有的变化进行了重放。）

二进制日志包含所有改变数据库的信息。请注意，那些尚没有但是可能改变数据库的语句也会被记录下来。最常见的可能会改变数据库的语句有：`DROP TABLE IF EXISTS` 或 `CREATE TABLE IF NOT EXISTS`，以及那些不匹配任何行的语句，如带有 `WHERE` 条件的 `DELETE` 和 `UPDATE` 语句。

`SELECT` 语句一般不会被记录，因为它们不会对数据库做任何改动。当然，也有例外。

二进制日志按照 master 上事务提交的顺序记录它们。尽管 master 上的事务可能是交错执行的，但每个事务在二进制日志中仍然是连续记录的，这取决于事务的提交时间。

## 二进制日志的结构

从概念上讲，二进制日志是一系列二进制日志事件（又称 binlog 事件，在不产生概念混淆的情况下也可以简称为事件）。从第 3 章可以知道，实际上，二进制日志包含若干个文件，如图 4-1 所示，它们一起构成了二进制日志。

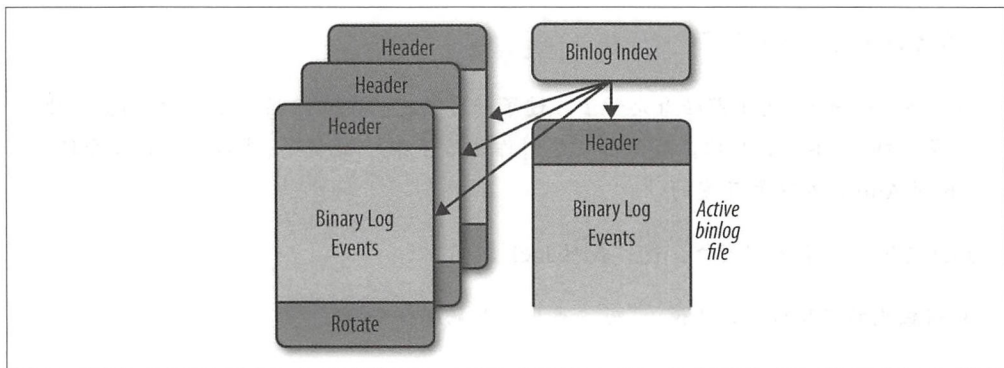


图4-1：二进制日志的结构

事件本身存储在一系列 binlog 文件中，文件名类似于 `host-bin.000001`；还有一个 binlog 索引文件，文件名通常为 `host-bin.index`，用来追踪已有的 binlog 文件。当前服务器正在写入的 binlog 文件称为活动（active）binlog 文件。如果所有 slave 都与 master 同步，那么 slave 也正在读取该文件。分别使用 `log-bin` 和 `log-bin-index` 选项来控制 binlog

文件及 binlog 索引文件的名称，这两个选项在第 3 章“配置 master”一节中提到过。稍后我们将进行详细讨论。

索引文件跟踪服务器上的所有 binlog 文件，以便必要的时候服务器能正确地创建新的 binlog 文件，哪怕是服务器重启以后。索引文件的每一行都包含一个二进制日志的 binlog 文件名。根据 MySQL 版本不同，这个文件名可以是一个完整路径或相对路径加文件名。影响 binlog 文件的命令同样也会影响索引文件，如 PURGE BINARY LOGS、RESET MASTER，以及 FLUSH LOGS 等命令，在增删 binlog 行的时候，也会导致匹配这些行的 binlog 文件更改（增加或删除）索引文件中的记录。

如图 4-1 所示，每个 binlog 文件由若干 binlog 事件组成，以 Format\_description 事件（格式描述事件）作为文件头，以 Rotate 事件（日志轮换事件）作为文件尾。注意，如果服务器突然停止或死机，binlog 文件末尾可能不是轮换事件。

Format\_description 事件包含写 binlog 文件的服务器信息，以及关于文件状态的关键信息。如果服务器关闭或重新启动，会创建一个新的 binlog 文件，同时向其中写入一个新的 Format\_description 事件。因为在服务器关闭和重启这段时间内可能产生更新，所以这个事件是必需的。例如，如果升级服务器，需要写入新的 Format\_description 事件。

服务器写完 binlog 文件后，在文件末尾添加一个 Rotate 事件。该事件指向下一个 binlog 文件，给定其文件名和读操作的初始位置。

Format\_description 事件和 Rotate 事件将在下一节进行详细描述。

除了控制事件（如 Format\_description 和 Rotate 事件），binlog 文件中的其他事件都被分成组（group），如图 4-2 所示。在事务存储引擎中，每个组大致对应一个事务；但是对于非事务存储引擎，或不属于某个事务的语句来说，例如 CREATE 或 ALTER 语句，每个语句本身就是一个组。<sup>注 1</sup>

总之，binlog 文件中的事件组要么是不属于事务的单个语句，要么是由多条语句组成的事务。

每个组要么全都执行，要么全都不执行（除了一些特殊情况）。如果由于某种原因 slave 在组执行的过程中停机，那么复制的起点是该组的起点，而不是停机时最后执行的语句。第 8 章将详细描述 slave 是如何执行事件的。

注 1 有些特殊情况下，非事务型语句也可能是某个组的一部分。这将在本章后面的“如何写入非事务型语句”一节中讲述。





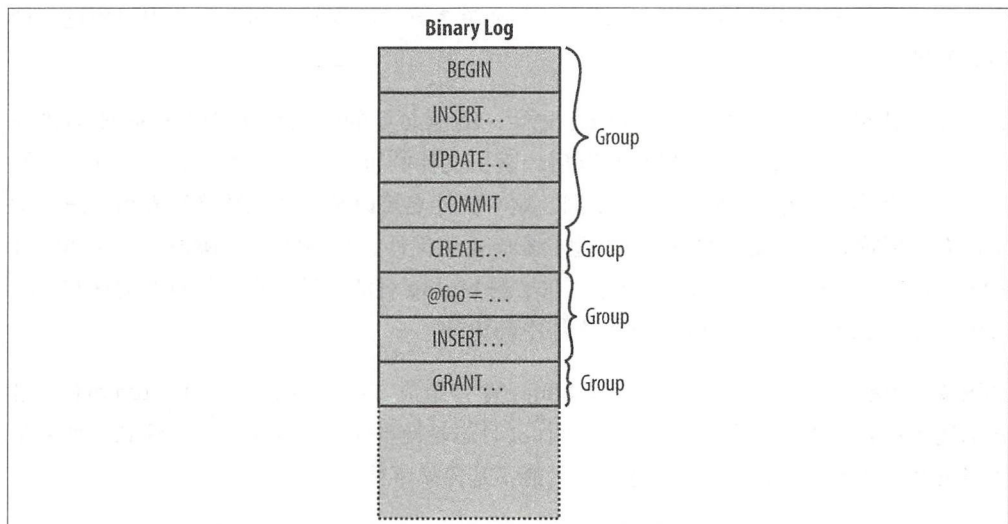


图4-2: 含多个事件组的单个binlog文件

## binlog 事件的结构

MySQL 5.0 引入了一个新的 binlog 格式，即二进制日志格式 4（binlog format 4）。如果需求增加，以前的格式很难添加新字段，所以二进制日志格式 4 是专门为可扩展设计的。尽管 5.0 以后的服务器扩展了 binlog 格式，添加了新事件，并为某些事件添加了新字段，binlog 格式仍然是服务器的事件格式。本章提到的事件格式即二进制日志格式 4。

每个 binlog 事件由 4 部分组成。

通用头（Common header）

通用头，顾名思义，就是 binlog 文件中所有事件通用的信息。

通用头包含事件的基本信息，其中最重要的字段是事件类型和事件的大小。

55 提交头（Post header）

提交头与特定的事件类型有关。也就是说，对不同的事件类型来说，该字段存储的信息不同。但是，与通用头一样，给定一个 binlog 文件，提交头的大小是相同的。事件类型的大小由 Format\_description 事件给出。

事件体（Event body）

事件头后面是事件体，其大小可变。事件的通用头中列出了事件体的大小和结束位置。事件体存储事件的主要数据，因事件类型不同而异。例如，Query 事件的事件体存储查询，而 User\_var 事件的事件体则存储某个语句中的用户变量名及其值。



## 校验和 (Checksum)

从 MySQL 5.6 开始, 如果服务器设置为产生校验和的话, 事件末尾就多了一个校验和字段。校验和是一个 32 位整型数, 用于检查事件写入后是否有损坏。

本书不讨论所有事件的完整格式列表, 但由于 `Format_description` 和 `Rotate` 事件对解释其他事件至关重要, 因此我们将在这里对它们做简要介绍。想了解更多事件的细节, 请参考 MySQL 内部手册。

前面讲过, `Format_description` 事件位于每个 binlog 文件的头部, 描述 binlog 文件中事件的公共信息。不同文件的 `Format_description` 事件可以不同; 这通常发生在服务器升级和重新启动的时候。

## binlog 文件格式版本

这是 binlog 文件的版本, 不要与服务器的版本混淆。MySQL 3.23, 4.0 和 4.1 版本使用的是二进制日志格式 3, 而 MySQL 5.0 和之后的版本使用的是二进制日志格式 4。

如果开发者对文件或事件的整体结构做了重大改动, 则 binlog 文件格式版本会发生变化。5.0 版本的 binlog 文件的起始事件采用了新的格式, 而且事件的通用头也不一样, 所以 binlog 文件格式版本也变了。

## 服务器版本

表示创建文件的服务器的版本字符串, 包括服务器的版本及特定的 build 信息。其格式通常分为三部分, 即版本号、连字符和其他 build 信息。例如, “5.5.10-debug-log” 表示 5.5.10 服务器的调试构建版本。

56

## 通用头的长度 (common header length)

该字段存储了通用头的长度, 不同 binlog 文件该字段的值不同。除了 `Format_description` 和 `Rotate` 事件 (固定通用头长度), 其他事件的通用头长度都是可变的。

## 提交头的长度 (post-header lengths)

binlog 文件中每个事件的提交头长度都是固定的, 该字段存储了各个事件的提交头长度构成的数组。由于不同服务器的事件数目不同, 所以这个字段前面还存储了服务器产生的事件类型数目。



`Rotate` 和 `Format_description` 日志事件的长度是固定的, 因为服务器在还不知道通用头长度之前就需要它们。在连接服务器的时候, 首先要发送 `Format_description` 事件。`Format_description` 事件中存储了通用头的长度, 如果这个长度不是固定值, 服务器就无从知道 `Rotate` 事件的通用头长度。因此, 这两个事件的通用头的长度都是固定的, 而且不会随着版本变化而改变, 哪怕其他事件的通用头变了, 它们也保持不变。



由于 `Format_description` 事件给出了各个事件类型的通用头大小和提交头大小，所以不管是添加新事件，还是增加新字段使得提交头的大小增加，都不会影响 binlog 文件的整体格式。

每次扩展格式都要特别小心，确保这些扩展不会影响早期版本中事件的解释。例如，向通用头添加一个字段表明该事件是否被压缩及其压缩类型。如果 slave 从旧版本的 master 上读取事件，就会没有这个字段，要确保服务器仍要能够以某种默认的方式处理这个事件。

## 事件校验

由于硬件可能失效，软件可能有错误，所以需要某种方法来保证损坏的数据不会被应用到 slave 上。任何时候都可能发生随机错误，如果某个语句内部出错，常常会产生句法错误，从而导致 slave 停止。然而，指望用这种方式来阻止损坏事件的复制、确保二进制日志的完整性，并不是一个好办法。这种策略无法捕获到确切的损坏类型，例如时间戳等，而且对于基于行的事件（即数据以二进制的形式编码）也不管用，实际上基于行的事件中的随机损坏更可能导致数据不准确。

为了保证每个事件的完整性，MySQL 5.6 引入了复制事件校验和。在写事件的时候，添加一个校验和；然后在读取这些事件的时候，计算这个校验和，并与之前写入的值进行比较。如果校验和不匹配，则在 slave 应用该事件之前终止执行。校验和的计算可能会影响性能。不过，基准测试已经证明添加和检查校验和并不会带来明显的性能下降。所以在 MySQL 5.6 中，默认启用事件校验功能。当然，必要的时候你也可以禁用它。

在 MySQL 5.6 中，当写二进制日志或中继日志的时候产生校验和，当从这些日志中读取事件的时候验证校验和。

通过以下三个选择控制复制事件校验和。

`binlog-checksum=type`

使用该选项启用校验和，并告诉服务器用何种方式计算校验和。目前有两种选择：

(1) CRC32，使用 ISO-3309 CRC-32 校验和；(2) NONE，即关闭校验和。默认值是 CRC32，即默认产生校验和。

`master-verify-checksum=boolean`

该选项表示读取二进制日志的时候 master 是否要验证校验和。也就是说，dump 线程（参见第 8 章的“复制架构基础”一节）从二进制日志中读取事件后，验证其校验和，无误后发送给 slave；同理使用 `SHOW BINLOG EVENTS` 命令也是。如果有任何损坏事件，命令就抛出一个错误。该选项默认处于关闭状态。



`slave-sql-verify-checksum=boolean`

该选项表示读取中继日志之后、在 slave 数据库上应用事件之前，slave 是否要验证事件的校验和。该选项默认处于关闭状态。

如果二进制日志或中继日志损坏，可以使用带有 `--verify-binlog-checksum` 选项的 `mysqlbinlog` 发现坏的校验和。该选项让 `mysqlbinlog` 验证每个事件的校验和，并在发现损坏事件的时候停止，例如：

```
$ client/mysqlbinlog --verify-binlog-checksum master-bin.000001
.
.
# at 261
#110406 8:35:28 server id 1 end_log_pos 333 CRC32 0xed927ef2...
SET TIMESTAMP=1302071728/*!*/;
BEGIN
/*!*/;
# at 333
#110406 8:35:28 server id 1 end_log_pos 365 CRC32 0x01ed254d Intvar
SET INSERT_ID=1/*!*/;
ERROR: Error in Log_event::read_log_event(): 'Event crc check failed!...
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

58

## 将语句写入日志

从 MySQL 5.1 开始，也支持基于行的复制（row-based replication），参见本章后面的“基于行的复制”一节。

在基于语句的复制（statement-based replication）中，实际执行的语句连同某些执行信息将一起被写入二进制日志，然后在 slave 上重新执行这些语句。当然并不是所有事件都会以语句的形式写入日志，要注意有些例外情况。本节讲述将单个语句写入日志的过程，并给出一些重要的说明。

由于二进制日志是公共资源，所有线程都要写二进制日志，所以一定要避免两个线程同时更新二进制日志。因此，在事件组写二进制日志之前，二进制日志将获得一个互斥锁 `LOCK_log`，然后在事件组写完后释放。由于服务器所有的会话线程都向二进制日志写事务，所以这个锁常常会阻塞某些会话线程。





## 写入 DML 语句

数据操作语言（DML）语句通常是指 DELETE、INSERT 和 UPDATE 语句。为了保证日志的一致性，MySQL 在写二进制日志的同时获取事务级锁，写完后释放锁。

在表上的锁释放前，在语句提交的同时需要将该语句写入二进制日志，这样就保证了二进制日志始终与语句对表的改动保持一致。如果记录日志没有作为语句的一部分时，有可能其他的语句会“插在”这条语句对数据库的修改和记录二进制日志这两个动作中间。也就是说，语句被写入日志的顺序，可能与数据库上的执行顺序不同，因此这样会导致 master 和 slave 之间不一致。例如，一个带有 WHERE 条件的 UPDATE 语句在 slave 上更新的行可能与 master 上不同，因为如果语句写入日志的顺序不同，更新的行可能也会不同。

## 写入 DDL 语句

数据定义语言（DDL）语句影响数据模式（schema），这些语句通常指 CREATE TABLE 和 ALTER TABLE 语句。DDL 语句会在文件系统中创建或改变对象，例如，表定义存储在 .frm 文件中，而数据库表现为文件系统中的目录。因此服务器需要将这些信息保存在内部数据结构中。为了保护这个内部数据结构的更新，在修改表定义之前需要先获得一个内部锁（称为 LOCK\_open）。

因为只有一个锁保护这些数据结构，所以数据库对象的创建、更新和销毁都可能带来性能问题，例如创建和销毁临时表——临时表是通过创建中间结果集来执行计算任务的常用技术。

如果创建和销毁大量的临时表，通常可以通过减少创建临时表的数目（从而也减少临时表的销毁）来改善性能。

## 写入查询

对于基于语句的复制来说，最常见的 binlog 事件是 Query 事件，它将 master 上执行的语句写入日志。除了实际执行的语句本身，该事件还包含执行语句必需的附加信息。

回想一下，二进制日志有很多用途，而且写入语句的顺序可能与 master 上语句执行的顺序不同。有时候可以向服务器回放二进制日志以执行 PITR，有时候可以从事件序列的中间位置开始复制（因为复制开始前 slave 已经进行了备份恢复）。

无论哪种情况，事件都在不同的上下文（context）中执行。上下文是指服务器执行语句时必须知道的隐式（implicit）信息，以保证语句能够正确执行。例如：





## 当前数据库

如果语句中引用了表、函数或过程，而没有指定哪个数据库，则默认使用当前数据库。

## 用户自定义变量的值

如果语句中引用了用户自定义的变量，则这个变量的值是隐式的。

## RAND 函数的种子

RAND 函数是基于伪随机数函数的函数，即生成一系列可再生的数字，看上去是随机的，但实际上是均匀分布的。该函数并不是真正随机的，而是从一个种子数字开始，然后应用一个伪随机函数来产生确定性的数字序列。也就是说，如果种子相同，RAND 函数总会返回相同的数字。所以，种子是隐式信息。

## 当前时间

显然，语句开始执行的时间是隐式信息。如果函数调用依赖于当前时间（如 NOW 和 UNIX\_TIMESTAMP 函数），那么保证时间的正确性就非常重要。如果 master 和 slave 有延迟，导致语句执行时间不一致，则语句将返回不同的结果。

## AUTO\_INCREMENT 字段的插入值

向表中插入行，如果该行含有 AUTO\_INCREMENT 类型的字段，那么对新插入的行来说，该字段值是隐式的，因为它与上一行有关。

## 调用 LAST\_INSERT\_ID 的返回值

如果语句调用了 LAST\_INSERT\_ID 函数，那么其值取决于上一个语句插入的值，因此也是隐式的。

## 线程 ID

有些语句的线程 ID 是隐式信息。例如，如果某个语句使用了临时表或调用了 CURRENT\_ID 函数，那么该语句的线程 ID 是隐式的。

无论在 slave 还是 master 上，宕机或重启后重放语句的时候，语句执行的上下文都是未知的，必须将隐式信息显式化，然后写入二进制日志。根据隐式信息的种类不同，其实现略有差异。

除了前面列出的隐式信息，还有一些信息对触发器和存储例程的执行是隐式的，我们将单独在本章后面的“触发器、事件和存储例程”一节中进行讲述。

下面我们逐个讨论每一种隐式信息，指出其存在的问题，以及服务器如何处理它。

## 当前数据库

向 Query 事件添加一个特殊字段，记录当前数据库。处理 LOAD DATA INFILE 语句的事件



61 中也有这个字段（详见本章后面的“LOAD DATA INFILE 语句”一节），所以这里的描述也适用于该语句。当前数据库还有一个重要用途，即数据库过滤，这将在本章后面进行介绍。

## 当前时间

有 5 个函数需要利用当前时间：NOW、CURDATE、CURTIME、UNIX\_TIMESTAMP 和 SYSDATE。前 4 个函数返回开始执行语句的时间，而 SYSDATE 将返回函数执行时的时间。我们在 NOW 和 SYSDATE 之间加入一个睡眠时间，通过比较发现它们之间的区别。

```
mysql> SELECT SYSDATE(), NOW(), SLEEP(2), SYSDATE(), NOW()\G
***** 1. row *****
SYSDATE(): 2013-06-08 23:24:08
NOW(): 2013-06-08 23:24:08
SLEEP(2): 0
SYSDATE(): 2013-06-08 23:24:10
NOW(): 2013-06-08 23:24:08
1 row in set (2.00 sec)
```

这两个函数都会计算时间值，但是 NOW 返回的是开始执行语句的时间，而 SYSDATE 返回的是执行函数的时间。

为了正确处理这些时间函数，事件将存储一个时间戳，表明事件何时开始执行。然后将这个时间戳的值从事件复制到 slave 执行进程，在计算时间函数的时候，将这个时间戳的值作为事件开始执行的时间。

由于 SYSDATE 直接从操作系统获取时间，这对基于语句的复制来说是不安全的，可能导致 master 和 slave 上执行的返回值不同。所以除非你真想把实际时间插入数据表，否则最好慎用这个函数。

## 上下文事件

有些语句也含有隐式信息，这些语句需要满足一些条件：

- 如果语句包含对用户定义变量的引用（如示例 4-1 中），就需要将用户定义变量的值写入二进制日志。
- 如果语句中包含 RAND 函数的调用，则需要将伪随机种子写入日志。
- 如果语句包含对 LAST\_INSERT\_ID 函数的调用，则需要将上一次插入的 ID 写入二进制日志。
- 如果语句需要向表中插入 AUTO\_INCREMENT 类型的字段，则需要将这个字段（或多个字段）的值写入二进制日志。

62



示例4-1: 包含用户定义变量的语句

```
SET @value = 45;  
INSERT INTO t1 VALUES (@value);
```

无论上述哪一种情况,在写入包含查询的事件之前,需要向二进制日志中写入一个或多个上下文事件(context events)。由于一个 Query 事件之前可能有多个上下文事件,所以二进制日志要同时处理多个用户定义变量和 RAND 函数,或者前面几种情况的(几乎)任意组合。二进制日志通过下面的事件来存储必要的上下文信息:

User\_var

该事件记录单个用户自定义变量的变量名及其值。

Rand

记录 RAND 函数的随机数的种子。种子取自会话内部状态。

Intvar

如果语句要插入 AUTO\_INCREMENT 类型的字段,在执行插入语句前,该事件会设置表中自动增量计数器的值。

如果语句包含 LAST\_INSERT\_ID 函数调用,该事件记录这个函数在该语句中的返回值。

示例 4-2 给出了产生上下文事件的语句示例,并使用 SHOW BINLOG EVENTS 命令展示这些事件。注意每个语句之前可能有多个上下文事件。

示例4-2: 带有上下文事件的查询事件

```
master> SET @foo = 12;  
Query OK, 0 rows affected (0.00 sec)
```

```
master> SET @bar = 'Smoothnoodlemaps';  
Query OK, 0 rows affected (0.00 sec)
```

```
master> INSERT INTO t1(b,c) VALUES  
-> (@foo,@bar), (RAND(), 'random');  
Query OK, 2 rows affected (0.00 sec)  
Records: 2 Duplicates: 0 Warnings: 0
```

```
master> INSERT INTO t1(b) VALUES (LAST_INSERT_ID());  
Query OK, 1 row affected (0.00 sec)
```

```
master> SHOW BINLOG EVENTS FROM 238\G
```

```
***** 1. row *****  
Log_name: mysqld1-bin.000001
```



```

        Pos: 238
Event_type: Query
  Server_id: 1
End_log_pos: 306
      Info: BEGIN
***** 2. row *****
      Log_name: mysql1-bin.000001
        Pos: 306
Event_type: Intvar
  Server_id: 1
End_log_pos: 334
      Info: INSERT_ID=1
***** 3. row *****
      Log_name: mysql1-bin.000001
        Pos: 334
Event_type: RAND
  Server_id: 1
End_log_pos: 369
      Info: rand_seed1=952494611,rand_seed2=949641547
***** 4. row *****
      Log_name: mysql1-bin.000001
        Pos: 369
Event_type: User var
  Server_id: 1
End_log_pos: 413
      Info: @`foo`=12
***** 5. row *****
      Log_name: mysql1-bin.000001
        Pos: 413
Event_type: User var
  Server_id: 1
End_log_pos: 465
      Info: @`bar`=_utf8 0x536D6F6F74686E6F6F6F...
***** 6. row *****
      Log_name: mysql1-bin.000001
        Pos: 465
Event_type: Query
  Server_id: 1
End_log_pos: 586
      Info: use `test`; INSERT INTO t1(b,c) VALUES (@foo,@bar)...
***** 7. row *****
      Log_name: mysql1-bin.000001
        Pos: 586
Event_type: Xid
  Server_id: 1

```

```

End_log_pos: 613
Info: COMMIT /* xid=44 */
***** 8. row *****
Log_name: mysql1-bin.000001
Pos: 613
Event_type: Query
Server_id: 1
End_log_pos: 681
Info: BEGIN
***** 9. row *****
Log_name: mysql1-bin.000001
Pos: 681
Event_type: Intvar
Server_id: 1
End_log_pos: 709
Info: LAST_INSERT_ID=1
***** 10. row *****
Log_name: mysql1-bin.000001
Pos: 709
Event_type: Intvar
Server_id: 1
End_log_pos: 737
Info: INSERT_ID=3
***** 11. row *****
Log_name: mysql1-bin.000001
Pos: 737
Event_type: Query
Server_id: 1
End_log_pos: 843
Info: use `test`; INSERT INTO t1(b) VALUES (LAST_INSERT_ID())
***** 12. row *****
Log_name: mysql1-bin.000001
Pos: 843
Event_type: Xid
Server_id: 1
End_log_pos: 870
Info: COMMIT /* xid=45 */
12 rows in set (0.00 sec)

```

## 线程 ID

二进制日志偶尔需要的最后一个隐式信息是处理语句的 MySQL 会话的线程 ID。如果调用了某些依赖于线程 ID 的函数，例如 `CONNECTION_ID`，就必须知道线程 ID。当然，在处理临时表时，线程 ID 额外重要。



临时表是依赖于线程的，也就是说，如果定义在不同的会话中，允许同时存在两个同名的临时表。临时表能够有效地提高某些操作的性能，但是相应的，二进制日志需要一些特殊处理工作。

服务器内部通过创建晦涩的表名来定义临时表。临时表的名字由服务器的进程 ID(process ID)、创建表的线程 ID(thread ID) 和一个线程计数器组成，该计数器用来区分同一个线程中不同的临时表实例。这种命名方式能够区分不同线程创建的表，但是只有当线程 ID 被存入二进制日志，语句才能访问相应的表。

与二进制日志处理当前数据库的方法类似，线程 ID 也作为一个独立字段存储在每个 Query 事件中，因此可以用线程 ID 字段来计算线程特定的数据，并正确处理临时表。

写 Query 事件时，其线程 ID 是从服务器变量 `pseudo_thread_id` 中读取的。也就是说，这个值可以在执行语句前设置，但前提是你有 SUPER 权限。这个服务器变量也可以用于 `mysqlbinlog` 工具以发出正确命令，但通常不那么用。

如果语句中含有 `CONNECTION_ID` 函数调用，或是使用或创建了临时表，那么该 Query 事件在二进制日志中被标记成“线程特定的”。而 Query 事件总是含有线程 ID，所以这个标记并不是必需的。其主要目的是避免 `mysqlbinlog` 中不必要的 `pseudo_thread_id` 变量赋值。

## LOAD DATA INFILE 语句

使用 `LOAD DATA INFILE` 语句很容易将文件中的数据快速读入表中。但遗憾的是，这依赖于特定的上下文信息，即从文件系统读取的文件，这种上下文与上文讨论的上下文事件不同。

为了处理 `LOAD DATA INFILE`，MySQL 服务器采用了一套特殊的事件，通过二进制日志进行文件传输。稍后你将看到，这样不仅解决了 `LOAD DATA INFILE` 的问题，还可以用该语句很方便地将大量数据从 master 传输到 slave。要正确传输数据和执行 `LOAD DATA INFILE` 语句，需要引入一些新的事件：

### Begin\_load\_query

这个事件告知数据传输在文件中的起点。

### Append\_block

如果这个文件超过连接数据包的最大容量，那么 `Begin_load_query` 事件后面的一个或多个 `Append_block` 事件构成的序列保存该文件的剩余数据。

## Execute\_load\_query

这个事件是 Query 事件的变种，保存 master 上执行的 LOAD DATA INFILE 语句。

该事件中的语句使用的文件位于 master 上，slave 无法找到这个文件。但可以通过前面的 Begin\_load\_query 和 Append\_block 事件获取文件内容。

对 master 上执行的每个 LOAD DATA INFILE 语句而言，要读取的文件被映射到一个内部文件缓冲区，以备使用。此外，该语句的执行还被分配一个唯一的文件 ID，表示该语句要读取的那个文件。

66

执行语句的时候，该文件内容以事件序列的形式写入二进制日志，该事件序列以 Begin\_load\_query 事件开头，表示新文件的开始，然后是零个或多个 Append\_block 事件。每个写入二进制日志的事件大小不能超过包的最大容量，这个最大值由 max\_allowed\_packet 选项指定。

当整个文件读取到表中后，向二进制日志写入 Execute\_load\_query 事件，语句的执行结束。这个事件包含了执行语句和分配给该语句的文件 ID。请注意，这个语句不是用户最开始写的那个语句，而是后来重新创建的变种。



如果你用的是老版本的二进制日志，则会有这几个事件：Load\_log\_event、Execute\_log\_event 和 Create\_file\_log\_event。MySQL 5.0.3 之前的版本使用这些事件复制 LOAD DATA INFILE，而新版本用的是上文描述的事件。

示例 4-3 展示了成功执行一个 LOAD DATA INFILE 语句写入二进制日志的事件信息。在这个例子中，从 Info 字段可以看出分配的文件 ID 是 1，而且语句执行的所有事件都要用到这个字段。另外，该语句所使用的文件 *foo.dat* 超过了最大数据包容量，即 16384，所以它被分成三个事件。

### 示例4-3：成功执行LOAD DATA INFILE

```
master> SHOW BINLOG EVENTS IN 'master-bin.000042' FROM 269\G
***** 1. row *****
Log_name:  master-bin.000042
Pos:      269
Event_type: Begin_load_query
Server_id: 1
End_log_pos: 16676
Info:      ;file_id=1;block_len=16384
***** 2. row *****
Log_name:  master-bin.000042
Pos:      16676
```

```

Event_type: Append_block
Server_id: 1
End_log_pos: 33083
Info: ;file_id=1;block_len=16384
***** 3. row *****
Log_name: master-bin.000042
Pos: 33083
Event_type: Append_block
Server_id: 1
End_log_pos: 33633
Info: ;file_id=1;block_len=527
***** 4. row *****
Log_name: master-bin.000042
Pos: 33633
Event_type: Execute_load_query
Server_id: 1
End_log_pos: 33756
Info: use `test`; LOAD DATA INFILE 'foo.dat' INTO...;file_id=1
4 rows in set (0.00 sec)

```

## 二进制日志过滤器

可以通过两个选项从二进制日志中过滤语句：binlog-do-db 和 binlog-ignore-db（我们统一称为 binlog-\*-db）。如果只是过滤特定数据库的语句使用 binlog-do-db，而如果想忽略某个数据库而复制其他所有数据库时则使用 binlog-ignore-db。

这些选项可以多次使用。如果要同时过滤 one\_db 数据库和 two\_db 数据库，你必须在 my.cnf 文件中同时设置这两个选项，例如：

```

[mysqld]
binlog-ignore-db=one_db
binlog-ignore-db=two_db

```

MySQL 过滤事件的方式会令不熟悉的用户相当吃惊，所以我们将解释过滤器是如何工作的，并给出一些避免问题的建议。

图 4-3 展示了 MySQL 如何确定是否过滤某条语句。这种过滤是在语句级完成的，也就是说，整个语句要么完全被过滤出去，要么完全被写入二进制日志。binlog-\*-db 选项使用 current 数据库来决定是否需要过滤语句，而不是受语句影响的表所在的数据库。

为了进一步理解这个行为，请参看示例 4-4，其中 bad 是当前数据库，而语句改变的数据表位于其他数据库。

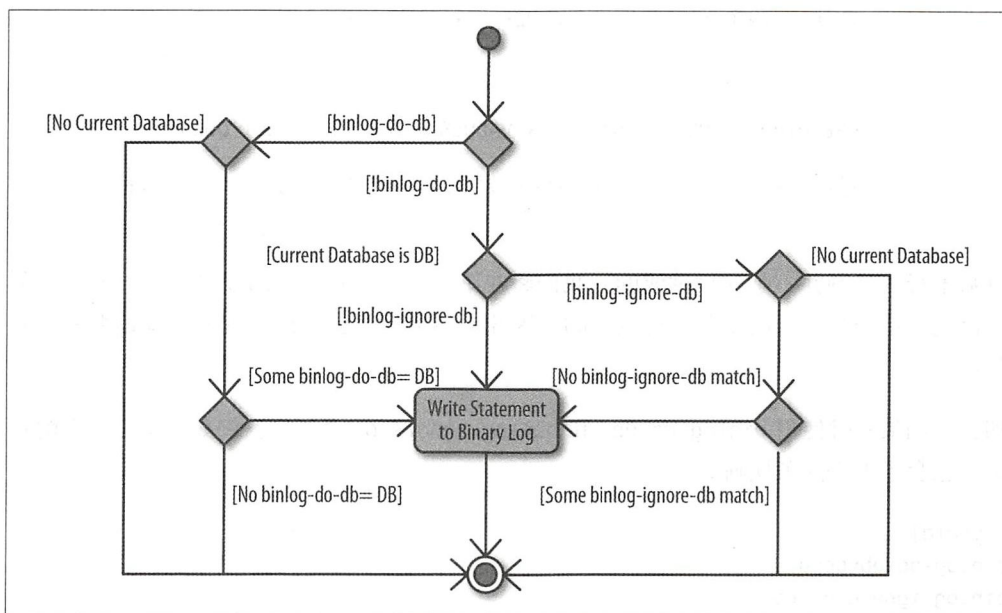


图4-3: binlog-*\**-db过滤器的逻辑

#### 示例4-4: 涉及不同数据库的语句

```
USE bad; INSERT INTO t1 VALUES (1),(2); ❶
USE bad; INSERT INTO good.t2 VALUES (1),(2); ❷
USE bad; UPDATE good.t1, ugly.t2 SET a = b; ❸
```

- ① 第一行更改了 bad 数据库中的一张表，因为表名前面没有指定数据库名。
- ② 第二行更改了其他数据库中的表。
- ③ 第三行更改了两个数据库中的两个表，这两个数据库都不是当前数据库。

现在，使用 `binlog-ignore-db=bad` 筛选 `bad` 数据库，看看会发生什么：示例 4-4 中的三个语句都不会被写入二进制日志，尽管第二个和第三个语句更改的是 `good` 和 `ugly` 数据库中的表。这乍看可能有点奇怪——为什么不根据表所在的数据库来过滤语句呢？假如根据 `ugly` 数据库而不是 `bad` 数据库过滤，我们看看第三个语句会怎样：`UPDATE` 语句中一个数据库被过滤了而另一个没有。这会使服务器陷入自我矛盾的状况（`catch-22`），而过滤当前数据库就可以解决这个问题。这个规定适用于所有语句（除少数例外）。



为了避免执行那些可能已经被过滤的语句而导致错误，写语句的时候要养成习惯，不要在表名、函数名或存储过程名前指定数据库名。每当你访问其他数据库中的表时，先用 USE 语句将那个数据库变成当前数据库。比如，不要这样写：



```
INSERT INTO other.book VALUES ('MySQL', 'Paul DuBois');
```

而应该这样写：

```
USE other; INSERT INTO book VALUES ('MySQL', 'Paul DuBois');
```

通过这种方式，数据表前面不指定数据库，语句就不会同时更新多个数据库。

但基于行的复制并不是这样过滤的，我们将在第 8 章的“基于行的复制中的过滤”一节中讨论。由于基于行的复制是行级别的，因此它不需要当前数据库就能过滤数据表中的某行。

那么，如果同时使用 `binlog-do-db` 和 `binlog-ignore-db` 会怎样？例如，假定一个配置文件包含了下面两条规则：

```
[mysqld]
binlog-do-db=good
binlog-ignore-db=bad
```

这种情况下，下面的语句到底会不会被过滤出去？

```
USE ugly; INSERT INTO t1 VALUES (1);
```

根据图 4-3 所示的逻辑，我们发现只要有至少一个 `binlog-do-db` 规则，则所有的 `binlog-ignore-db` 规则都可以完全被忽略。由于这里只保留 `good` 数据库，所以上面的这个语句将被过滤出去。



鉴于上述 `binlog-*-db` 规则的评估方式，同时设置 `binlog-do-db` 和 `binlog-ignore-db` 规则是毫无意义的。由于二进制日志将用来恢复和复制，我们不推荐使用 `binlog-*-db` 选项，而是使用 `replicate-*` 在 `slave` 上滤掉事件（参见第 8 章的“过滤和跳过事件”一节）。使用 `binlog-*-db` 选项会把语句从二进制日志过滤出去，这样一旦发生崩溃就无法从二进制日志恢复数据库。

## 触发器、事件和存储例程

记录日志时还有一些需要特别处理的结构：存储程序（stored programs），即触发器、事件和存储例程（stored routines，存储例程是存储过程和存储函数的总称）。二进制日志在处理它们的时候有一些共性，所以放在本节一起讨论。这里我们将语句分为两种类型：（1）定义、销毁或更改存储程序的语句，和（2）调用它们的语句。



## 定义或销毁存储程序的语句

下面我们将举例讨论触发器，同样的原理也适用于事件和存储例程。示例 4-5 中的代码解释了为什么在写二进制日志的时候服务器需要对它们给予特别处理。

在这个例子中，employee 表存储了所有员工信息，log 表保存了一些有趣的日志信息。注意，log 表中的 timestamp 字段记录更新时间，employee 表的主键是 name 字段，status 字段标记添加操作成功还是失败。

为了跟踪员工信息的变更（比如为了审计目的），为员工的添加、删除或更改操作一共创建了三个触发器，每次更新会向 log 表添加一条记录。

请注意这些触发器都是 after 触发器，即只有语句成功执行才会写 log 表。失败的语句不会被写入。稍后我们将举个扩展的例子，其中失败的语句也会被记录。

示例4-5：定义员工管理的表和触发器

```
CREATE TABLE employee (
    name CHAR(64) NOT NULL,
    email CHAR(64),
    password CHAR(64),
    PRIMARY KEY (name)
);

CREATE TABLE log (
    id INT AUTO_INCREMENT,
    email CHAR(64),
    status CHAR(10),
    message TEXT,
    ts TIMESTAMP,
    PRIMARY KEY (id)
);

CREATE TRIGGER tr_employee_insert_after AFTER INSERT ON employee
FOR EACH ROW
    INSERT INTO log(email, status, message)
    VALUES (NEW.email, 'OK', CONCAT('Adding employee ', NEW.name));

CREATE TRIGGER tr_employee_delete_after AFTER DELETE ON employee
FOR EACH ROW
    INSERT INTO log(email, status, message)
    VALUES (OLD.email, 'OK', 'Removing employee');

delimiter $$
CREATE TRIGGER tr_employee_update_after AFTER UPDATE ON employee
```

```

FOR EACH ROW
BEGIN
    IF OLD.name != NEW.name THEN
        INSERT INTO log(email, status, message)
            VALUES (OLD.email, 'OK',
                CONCAT('Name change from ', OLD.name, ' to ', NEW.name));
    END IF;
    IF OLD.password != NEW.password THEN
        INSERT INTO log(email, status, message)
            VALUES (OLD.email, 'OK', 'Password change');
    END IF;
    IF OLD.email != NEW.email THEN
        INSERT INTO log(email, status, message)
            VALUES (OLD.email, 'OK', CONCAT('E-mail change to ', NEW.email));
    END IF;
END $$
delimiter ;

```

定义完触发器以后，就可以按照示例 4-6 添加和删除员工了。如你所见，员工被添加、修改和删除，每个操作都被写入 log 表。

具有 employee 表访问权限的用户可以添加、删除和修改员工，那 log 表谁能访问呢？这里能够操纵 employee 表的用户应该不能更改 log 表。这有许多原因，但归根结底还是 log 表内容的信任问题，为了维护、审计、合法权威披露等目的。因此，DBA(数据库管理员)可以在允许多用户访问 employee 表的同时，严格控制 log 表的访问权限。

示例4-6: 添加、删除和修改用户

```

master> SET @pass = PASSWORD('xyzzy');
Query OK, 0 rows affected (0.00 sec)

```

```

master> INSERT INTO employee VALUES ('mats', 'mats@example.com', @pass);

Query OK, 1 row affected (0.00 sec)

```

```

master> UPDATE employee SET name = 'matz'
-> WHERE email = 'mats@example.com';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

```

```

master> SET @pass = PASSWORD('foobar');
Query OK, 0 rows affected (0.00 sec)

```

```

master> UPDATE employee SET password = @pass
-> WHERE email = 'mats@example.com';

```

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
master> DELETE FROM employee WHERE email = 'mats@example.com';
Query OK, 1 row affected (0.00 sec)
```

```
master> SELECT * FROM log;
```

id	email	message	ts
1	mats@example.com	Adding employee mats	2012-11-14 18:56:08
2	mats@example.com	Name change from mats to matz	2012-11-14 18:56:11
3	mats@example.com	Password change	2012-11-14 18:56:41
4	mats@example.com	Removing employee	2012-11-14 18:57:11

```
4 rows in set (0.00 sec)
```

在本例中，INSERT、UPDATE 和 DELETE 会警告该语句模式的日志是不安全的，因为它触发了一个插入 autoincrement 字段的触发器。通常这种警告应该要审查。

为了确保触发器可以成功地在一个被高度保护的表上执行，触发器通常由定义它们的用户执行，而不是更改 employee 表内容的用户来执行。因此，示例 4-5 中的 CREATE TRIGGER 语句由 DBA 执行，它同时具备更改 log 表的权限；而示例 4-6 中修改员工信息则是由一个用户管理账号执行，它仅具备修改 employee 表的权限。

在执行示例 4-6 中的语句时，员工管理账户用于更新 employee 表中的记录，而 DBA 权限则用于向 log 表添加记录。员工管理账户不能向 log 表添加或删除记录。

顺便说一句，在示例 4-6 中，用户变量在使用前被分配了密码。这样做是为了避免将敏感数据以纯文本形式发送给另一台服务器。

## 安全和二进制日志

73

一般来说，一个有 REPLICATION SLAVE 权限的用户能够读取 master 上的任何东西，因此应使保证账户的安全性。本书并不会详细讨论这个问题，只是列举一些预防措施的例子：

- 保证从防火墙外无法登录该账户。
- 跟踪所有具备 REPLICATION SLAVE 权限的账号的登录日志，并将其单独放在一个安全的服务器上。
- 加密 master 和 slave 间的连接，例如 MySQL 内置的 SSL (Secure Sockets Layer)。

虽然账户安全了，但二进制日志中可能还有一些不必要的信息，那么干脆一开始就不要保存它们。

密码是一种比较常见的敏感信息。如果访问表需要密码，当服务器在执行更改表的语句时，包含密码的事件会被写入二进制日志。

一个典型的例子是：

```
UPDATE employee SET pass = PASSWORD('foobar')
WHERE email = 'chuck@example.com';
```

如果要做复制的话，最好重写这个语句，把密码去掉。可以通过以下方法实现：将密码哈希计算后保存到一个用户定义变量，然后在表达式中使用这个变量：

```
SET @password = PASSWORD('foobar');
UPDATE employee SET pass = @password WHERE email = 'chuck@example.com';
```

由于 SET 语句不会被复制，原始密码将不会写入二进制日志，而仅存在于执行语句的服务器内存中。

只需要将哈希过的密码保存到表中，而不需密码原文，这种方法是有效的。如果要直接保存密码原文，就无法阻止密码被写入二进制日志。存储哈希密码在任何时候都是一个标准的好办法，防止有人从原始数据截获密码。

加密 master 和 slave 之间的连接提供了一些保护，但如果二进制日志本身被攻破，加密连接也无能为力。

回忆一下前面讨论过的隐式信息，你会发现，不管是执行某行代码的用户，还是定义触发器的用户，它们都是隐式的。在第 8 章你将看到，在 slave 上执行触发器的时候，是定义者还是调用者并不重要。但是，如果要把二进制日志重放到服务器（例如做即时恢复的时候），这就变得重要了。

要想在重放二进制日志的时候顺利处理各种表的权限问题，需要由具备 SUPER 权限的用户来执行所有语句。但是触发器可能不是 SUPER 权限的用户定义的，所以一定要用正确的用户重新定义和创建触发器。如果最初定义的触发器被具有 SUPER 权限的用户重新定义，会导致权限升级。

带有 DEFINER 从句的 CREATE TRIGGER 命令允许 DBA 指定触发器将在哪个用户下执行。如果没有指定 DEFINER（参见示例 4-7），在写二进制日志的时候该语句会被重写，自动为其添加定义者为当前用户的 DEFINER 从句。也就是说，insert 触发器的定义将出现在二进制日志中，如示例 4-7 所示，创建触发器的用户（即 root@localhost）为定义者。



#### 示例4-7：二进制日志中的CREATE TRIGGER语句

```
master> SHOW BINLOG EVENTS FROM 92236 LIMIT 1\G
```

```

***** 1. row *****
Log_name:  master-bin.000038
Pos:      92236
Event_type: Query
Server_id: 1
End_log_pos: 92491
Info:  use `test`; CREATE DEFINER=`root`@`localhost` TRIGGER ...
1 row in set (0.00 sec)

```

### 调用触发器和存储例程的语句

从定义谈到调用，大家可能会问，复制时如何处理 master 上的触发器呢。好吧，实际上它们根本不需要处理。

调用触发器的语句被写入二进制日志，但并没有关联特定的触发器。当 slave 执行这个语句的时候，它会自动执行相关表上的所有触发器。也就是说，master 和 slave 上可以存在不同的触发器，master 上的触发器在 master 上被调用，而 slave 上的触发器在 slave 上被调用。例如，如果 slave 不需要向 log 表添加记录的触发器，那么可以删除 slave 上的这个触发器以提高性能。

对于调用触发器的语句来说，语句之前任何复制必需的上下文事件都会写入二进制日志，包括那些触发器里面的需要上下文事件的语句。示例 4-8 显示了示例 4-5 中的 INSERT 语句执行后的二进制日志。注意，第一个事件用 log 表的主键写 INSERT ID，这是 log 表在触发器中的用处，但这好像是多余的，因为 slave 并不会使用该触发器。

75

尽管如此，还是要注意，在 master 和 slave 上使用不同的触发器（或者 master 和 slave 上根本就没有触发器）是特殊情况，如果某个触发器同时存在于 master 和 slave 上，复制 INSERT 语句就需要 INSERT ID。

#### 示例4-8：在执行INSERT之后二进制日志的内容

```
master> SHOW BINLOG EVENTS FROM 93340\G
```

```

***** 1. row *****
Log_name:  master-bin.000038
Pos:      93340
Event_type: Intvar
Server_id: 1
End_log_pos: 93368
Info:  INSERT_ID=1

```



```

***** 2. row *****
Log_name: master-bin.000038
Pos: 93368
Event_type: User var
Server_id: 1
End_log_pos: 93396h
Info: @`pass`=_utf8 0x2A394235303333433424335324...
utf8_general_ci
***** 3. row *****
Log_name: master-bin.000038
Pos: 93396
Event_type: Query
Server_id: 1
End_log_pos: 93537
Info: use `test`; INSERT INTO employee VALUES ...
3 rows in set (0.00 sec)

```

## 存储过程

存储函数 (stored function) 和存储过程 (stored procedure) 统称为存储例程 (stored routine)。由于服务器对存储过程和存储函数的处理是截然不同的，这一节我们讲述存储过程，而存储函数将在下一小节介绍。

存储例程和触发器在某些方面很类似，但有些地方却完全不同。与触发器类似的是，存储例程里面的 DEFINER 从句，无论语句本身是否有 DEFINER 从句，都会被显式写入二进制日志中。但是存储例程的调用方式与触发器不同。

**76** 示例 4-6 中定义了员工表和日志表，我们扩展一些实用例程用于员工。当然也可以使用 INSERT、DELETE 和 UPDATE 语句来实现，但这里我们使用存储过程来处理，来讲解将存储过程写入二进制日志涉及的一些问题。扩展示例见示例 4-9，我们添加了一些增加和删除员工的函数。

示例4-9：管理员工的存储过程定义

```

delimiter $$
CREATE PROCEDURE employee_add(p_name CHAR(64), p_email CHAR(64),
                             p_password CHAR(64))
MODIFIES SQL DATA
BEGIN
    DECLARE l_pass CHAR(64);
    SET l_pass = PASSWORD(p_password);
    INSERT INTO employee(name, email, password)
    VALUES (p_name, p_email, l_pass);

```

```

END $$

CREATE PROCEDURE employee_passwd(p_email CHAR(64),
                                p_password CHAR(64))
    MODIFIES SQL DATA
BEGIN
    DECLARE l_pass CHAR(64);
    SET l_pass = PASSWORD(p_password);
    UPDATE employee SET password = l_pass
    WHERE email = p_email;
END $$

CREATE PROCEDURE employee_del(p_name CHAR(64))
    MODIFIES SQL DATA
BEGIN
    DELETE FROM employee WHERE name = p_name;
END $$
delimiter ;

```

按照前面讲过的理论，对 `employee_add` 和 `employee_passwd` 过程，我们把加密后的密码提取到一个单独的变量中，而由于 `employee_del` 过程只有一个 `DELETE` 语句，所以不需要特殊处理。对应一个函数的一条 binlog 记录是这样的：

```

master> SHOW BINLOG EVENTS FROM 97911 LIMIT 1\G
***** 1. row *****
Log_name:  master-bin.000038
Pos:      97911
Event_type: Query
Server_id: 1
End_log_pos: 98275
Info:  use `test`; CREATE DEFINER=`root`@`localhost` PROCEDURE ...
1 row in set (0.00 sec)

```

我们看到，二进制日志在写过程定义的时候添加了 `DEFINER` 从句，除此之外，过程的主体部分不动。注意，同所有 DDL 语句一样，`CREATE PROCEDURE` 语句被作为一个 Query 事件来复制。

从这方面来讲，二进制日志对存储例程与触发器的处理方式是类似的。但它们的调用就大不相同了。示例 4-10 调用了添加员工的过程，并给出了相应的二进制日志内容。

示例4-10：调用存储过程

```

master> CALL employee_add('chuck', 'chuck@example.com', 'abrakadabra');
Query OK, 1 row affected (0.00 sec)

```

```
master> SHOW BINLOG EVENTS FROM 104033\G
```

```
***** 1. row *****
Log_name:  master-bin.000038
Pos:      104033
Event_type: Intvar
Server_id: 1
End_log_pos: 104061
Info:  INSERT_ID=1
***** 2. row *****
Log_name:  master-bin.000038
Pos:      104061
Event_type: Query
Server_id: 1
End_log_pos: 104416
Info:  use `test`; INSERT INTO employee(name, email, password)
      VALUES ( NAME_CONST('p_name',_utf8'chuck' COLLATE ...),
      NAME_CONST('p_email',_utf8'chuck@example.com' COLLATE ...),
      NAME_CONST('pass',_utf8'*FEB349C4FDAA307A...' COLLATE ...))
2 rows in set (0.00 sec)
```

在示例 4-10 中，有 4 件事需要注意：

- CALL 语句并没有被写入二进制日志，而这个调用的结果即存储过程中要执行的语句被写入二进制日志。换句话说，存储过程的主体被分解展开然后写入二进制日志。
- 语句被重写，去掉了所有引用存储过程的参数，即 p\_name、p\_email 和 p\_password。重写采用 NAME\_CONST 函数为每个参数创建一个单值的结果集。
- 局部声明变量 pass 也被换成了 NAME\_CONST 表达式，其中第二个参数是加密后的密码。
- 在向二进制日志写入调用触发器的语句时，首先写入一个包含 insert ID（向 log 表添加员工时使用）的 Intvar 事件，然后才写调用存储过程的语句。

由于参数名和局部声明变量名在存储例程外不可用，我们使用 NAME\_CONST 在执行函数的时候将参数和局部声明变量关联到某个常量值，这个值可以作为参数或局部变量使用。但是这么做改善并不明显，目前来说并不比直接使用参数更有优势。

## 存储函数

存储函数与存储过程有很多相同点，和触发器也有一些类似。与存储过程和触发器类似，存储函数也有一个 DEFINER 从句，通常（但不总是）用于将 CREATE FUNCTION 语句写入二进制日志。

与存储过程不同，存储函数可以返回标量值，所以可以在 SQL 语句中的各个地方嵌入使用。例如示例 4-11 中定义了一个存储例程，给定员工的姓名，提取员工的邮件地址。这个函数有点儿做作，直接执行语句更有效一些，但是它能说明我们的目的。

示例 4-11：一个获取员工姓名的存储函数

```
delimiter $$
CREATE FUNCTION employee_email(p_name CHAR(64))
  RETURNS CHAR(64)
  DETERMINISTIC
BEGIN
  DECLARE l_email CHAR(64);
  SELECT email INTO l_email FROM employee WHERE name = p_name;
  RETURN l_email;
END $$
delimiter ;
```

这个存储函数可以方便地被其他语句使用，如示例 4-12 所示。与存储过程不同，如果要写入二进制日志，存储函数必须指定一个特征，如 DETERMINISTIC、NO SQL 或 READS SQL DATA。

示例4-12：使用存储函数的例子

```
master> CREATE TABLE collected (
  -> name CHAR(32),
  -> email CHAR(64)
  -> );
Query OK, 0 rows affected (0.09 sec)

master> INSERT INTO collected(name, email)
  -> VALUES ('chuck', employee_email('chuck'));
Query OK, 1 row affected (0.01 sec)
```

```
master> SELECT employee_email('chuck');
+-----+
| employee_email('chuck') |
+-----+
| chuck@example.com      |
+-----+
1 row in set (0.00 sec)
```

在调用的时候，存储函数的复制方式与触发器相同，即作为执行函数的语句的一部分。例如，在示例 4-12 中，INSERT 语句之前二进制日志不写入任何事件，但是要写入 INSERT 内的存储函数要复制所需的上下文事件。

那 SELECT 呢？一般 SELECT 语句不会被写入二进制日志，因为它们不会改变任何数据。

但是如果 SELECT 语句含有存储函数就不同了，如示例 4-13 所示。

#### 示例4-13：更新表的存储函数示例

```
CREATE TABLE log(log_id INT AUTO_INCREMENT PRIMARY KEY, msg TEXT);
```

```
delimiter $$
CREATE FUNCTION log_message(msg TEXT)
  RETURNS INT
  DETERMINISTIC
BEGIN
  INSERT INTO log(msg) VALUES(msg);
  RETURN LAST_INSERT_ID();
END $$
delimiter ;
```

```
SELECT log_message('Just a test');
```

在执行这个存储函数的时候，服务器注意到 log 表增加了一行，于是将该语句标记为“更新”语句，这意味着它会被写入二进制日志。所以，对示例 4-13 来说，二进制日志会包含这些事件：

```
***** 7. row *****
```

```
Log_name: mysql-bin.000001
Pos: 845
Event_type: Query
Server_id: 1
End_log_pos: 913
Info: BEGIN
```

```
80 ***** 8. row *****
```

```
Log_name: mysql-bin.000001
Pos: 913
Event_type: Intvar
Server_id: 1
End_log_pos: 941
Info: LAST_INSERT_ID=1
```

```
***** 9. row *****
```

```
Log_name: mysql-bin.000001
Pos: 941
Event_type: Intvar
Server_id: 1
End_log_pos: 969
Info: INSERT_ID=1
```

```
***** 10. row *****
```

```
Log_name: mysql-bin.000001
Pos: 969
```



```

Event_type: Query
Server_id: 1
End_log_pos: 1109
Info: use `test`; SELECT `test`.`log_message`(_utf8'Just a test' COLLATE...
***** 11. row *****
Log_name: mysql-bin.000001
Pos: 1105
Event_type: Xid
Server_id: 1
End_log_pos: 1132
Info: COMMIT /* xid=237 */

```

## 存储函数和权限

定义存储过程或存储函数时需要 CREATE ROUTINE 权限。严格来讲，创建存储例程并不需要其他权限，但由于它通常需在定义者的权限下执行，所以如果定义者不具备读写相应表的权限，那么定义这个存储过程是没有意义的。

但是 slave 上的复制线程不需要权限检查即可执行。这是一个严重的安全漏洞，任何具备 CREATE ROUTINE 权限的用户都可以提升权限，在 slave 上执行任何语句。

在 MySQL 5.0 之前的版本中，这样并没有问题，因为 master 上的语句执行时会检查所有路径。一旦权限冲突，语句就不会写入二进制日志，因此用户无法在 slave 上访问到那些在 master 上被禁止的对象。然而，随着存储例程的引入，可能产生条件性执行路径，这样服务器在执行存储例程时无法检查所有路径。

由于存储过程被展开写入二进制日志，在 master 上执行的语句同样也会在 slave 上执行。而只有在 master 上成功执行的语句才会被写入日志，无从访问其他对象。但是存储函数不是这样的。

如果存储函数使用 SQL SECURITY INVOKER 定义，恶意用户可以精心设计一个在 master 和 slave 上的执行方式不同的函数。在 slave 上运行的时候并不知道这个安全漏洞。比如下面的例子：

```

CREATE FUNCTION magic()
  RETURNS CHAR(64)
  SQL SECURITY INVOKER
BEGIN
  DECLARE result CHAR(64);
  IF @@server_id <> 1 THEN
    SELECT what INTO result FROM secret.agents LIMIT 1;

```

◀ 81

```

    RETURN result;
ELSE
    RETURN 'I am magic!';
END IF;
END $$

```

其中一部分代码在 master 上执行 (ELSE 分支), 而有一段代码则在 slave 上执行 (IF 分支)。由于 slave 上没有权限检查, 结果相当于用户权限从 CREATE ROUTINE 提升为 SUPER。

注意, 如果函数定义中含有 SQL SECURITY DEFINER, 就不会有这个问题, 因为 slave 会拦截要求用户权限的函数执行。

为了防止 slave 上的权限提升问题, MySQL 中定义存储函数默认需要 SUPER 权限。不过由于存储函数非常有用, 而且一些数据库管理员相信用户能够正确创建函数, 因此可以通过 log-bin-trust-function-creators 选项禁用权限检查。

## 事件

事件是 MySQL 的扩展功能, 算不上标准 SQL。与 binlog 事件不同, 事件以存储程序的形式通过一个特殊的事件调度器定期执行。

同其他存储程序一样, 事件的定义连同 DEFINER 从句一起被写入日志。由于事件由事件调度器调用, 因此它们总是由定义者执行, 不存在存储函数那样的安全风险。

执行事件的时候, 语句被直接写入二进制日志。

由于事件在 master 上执行, 在 slave 上被自动禁用, 因此不会在 slave 上执行。如果没有禁用, 事件将在 slave 上执行两次: 一次是由 master 执行, 然后更新被复制到 slave, 还有一次是 slave 直接执行这个事件。



由于事件在 slave 上被禁用, 倘若出于某些原因 slave 丢失了 master 的连接, 就需要启用这些事件。

举个例子, 升级 slave (见第 5 章) 的时候, 不要忘了启用那些从 master 复制过来的事件。最简单的办法是使用下面的语句:

```

UPDATE mysql.events
    SET Status = ENABLED
    WHERE Status = SLAVESIDE_DISABLED;

```

确保只启用那些因复制被禁用的事件。可能有些事件是因为其他原因被禁用的。

## 特殊结构

尽管基于语句的复制通常比较简单，但有些特殊结构还是要小心处理。回想一下，要想在 slave 上正确执行语句，语句的上下文一定要正确。尽管前面讨论的上下文事件解决了一些上下文问题，但有些结构的上下文信息并没有在复制过程中传递。

### LOAD\_FILE 函数

LOAD\_FILE 函数可以读取某个文件，并将其使用在表达式中。有时候这很方便，但该函数要求 slave 上必须要有这个文件，因为复制过程并不会传递这个文件，而 LOAD DATA INFILE 中的文件则会被传递。所以，你可以用 LOAD DATA INFILE 语句巧妙地重写带有 LOAD\_FILE 函数的语句，或者自定义一个变量保存该文件的内容。例如，下面的语句将文档插入数据表中：

```
master> INSERT INTO document(author, body)
-> VALUES ('Mats Kindahl', LOAD_FILE('go_intro.xml'));
```

也可以用 LOAD DATA INFILE 重写上面的语句。这个时候，要注意指定的字段分隔符和行分隔符不能出现在文档中，因为我们要把整个文件的内容读入单个字段中。

```
master> LOAD DATA INFILE 'go_intro.xml' INTO TABLE document
-> FIELDS TERMINATED BY '@*@" LINES TERMINATED BY '&%&'
-> (author, body) SET author = 'Mats Kindahl';
```

还有一个办法就是把文件内容保存到某个用户定义变量中，然后在语句中使用这个变量。

```
master> SET @document = LOAD_FILE('go_intro.xml');
master> INSERT INTO document(author, body) VALUES
-> ('Mats Kindahl, @document);
```

83

## 非事务型变更和错误处理

到目前为止，我们只讨论了事务型变更，而且完全没有考虑错误处理。对事务型变更来说，错误处理并不复杂：如果语句尝试更改事务表失败，不会对这个表产生任何影响。这就是事务型系统的关键——语句尝试做的更改都会被忽略。回滚事务也是这样：已回滚的事务对表没有任何影响，所以直接被丢弃，而不会导致 master 和 slave 不一致。

MySQL 的特点之一是提供非事务型存储引擎。这可以带来一些速度上的优势，因为这个存储引擎不需要像事务型引擎那样管理事务日志，而且还能优化磁盘访问。但是，从复制的角度来说，非事务型引擎需要特殊处理。

需要特别注意的一个很重要的问题是，复制不能处理任意的非事务型引擎，必须要假设

非事务型引擎的工作方式。随着在 5.1 版本中引入了基于行的复制,某些限制被取消了(参见本章后面“基于行的复制”一节),但即便如此,还是不能处理任意的存储引擎。

从复制的角度看,有一个特性使问题更加复杂,即允许在同一事务中甚至是在同一个语句中混合使用事务型和非事务型引擎。

继续使用前面的例子,在示例 4-14 中,log 表是在非事务型存储引擎下创建的,而 employee 表是在事务型存储引擎下创建的。对于 log 表我们采用非事务型 MyISAM 存储引擎来提高速度,而 employee 表保持事务型行为。

进一步扩展这个例子,创建一对 insert 触发器追踪增加员工操作的失败情况: before 触发器和 after 触发器。如果管理员看到日志中有一条记录的状态字段为 FAIL,说明 before 触发器被调用而 after 触发器没有运行,也就是说增加员工操作失败。

示例4-14: 用存储引擎定义log表和employee表

```
CREATE TABLE employee (  
    name CHAR(64) NOT NULL,  
    email CHAR(64),  
    password CHAR(64),  
    PRIMARY KEY (email)
```

84 ) ENGINE = InnoDB;

```
CREATE TABLE log (  
    id INT AUTO_INCREMENT,  
    email CHAR(64),  
    message TEXT,  
    status ENUM('FAIL', 'OK') DEFAULT 'FAIL',  
    ts TIMESTAMP,  
    PRIMARY KEY (id)  
) ENGINE = MyISAM;
```

```
delimiter $$
```

```
CREATE TRIGGER tr_employee_insert_before BEFORE INSERT ON employee  
FOR EACH ROW  
BEGIN
```

```
    INSERT INTO log(email, message)  
        VALUES (NEW.email, CONCAT('Adding employee ', NEW.name));  
    SET @LAST_INSERT_ID = LAST_INSERT_ID();
```

```
END $$
```

```
delimiter ;
```

```
CREATE TRIGGER tr_employee_insert_after AFTER INSERT ON employee
FOR EACH ROW
    UPDATE log SET status = 'OK' WHERE id = @LAST_INSERT_ID;
```

这对二进制日志有什么影响呢？

首先，让我们考虑一下示例 4-6 中的 INSERT 语句。假设这个语句不属于某个事务中，且 AUTOCOMMIT 值为 1，那么这个语句本身就是一个事务。如果语句无错地执行，一切都将按计划进行，并且该语句将作为一个 Query 事件写入二进制日志。

现在，我们考虑如果重复执行同一个员工的 INSERT 语句会发生什么。因为 email 字段是主键，当尝试插入的时候会产生重复键的错误，但是这个语句呢？它会不会写入二进制日志？

让我们来看一看：

```
master> SET @pass = PASSWORD('xyzyz');
Query OK, 0 rows affected (0.00 sec)
```

```
master> INSERT INTO employee(name,email,password)
-> VALUES ('chuck','chuck@example.com',@pass);
ERROR 1062 (23000): Duplicate entry 'chuck@example.com' for key 'PRIMARY'
master> SELECT * FROM employee;
+-----+-----+-----+
| name | email | password |
+-----+-----+-----+
| chuck | chuck@example.com | *151AF6B8C3A6AA09CFCCBD34601F2D309ED54888 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
master> SHOW BINLOG EVENTS FROM 38493\G
```

```
***** 1. row *****
Log_name: master-bin.000038
Pos: 38493
Event_type: User var
Server_id: 1
End_log_pos: 38571
Info: @`pass`=_utf8 0x2A31353141463642384333413641413...
***** 2. row *****
Log_name: master-bin.000038
Pos: 38571
Event_type: Query
Server_id: 1
```



```

End_log_pos: 38689
Info: use `test`; INSERT INTO employee(name,email,password)...
2 rows in set (0.00 sec)

```

我们看到，尽管 `employee` 表是事务型的，而且语句执行失败了，这个语句还是被写入了二进制日志。用 `SELECT` 语句查看表的内容，我们发现只有一个 `employee`，说明这个语句已经被回滚了。那么，为什么这个语句还会写入二进制日志呢？

查看 `log` 表将发现原因。

```

master> SELECT * FROM log;
+-----+-----+-----+-----+-----+
| id | email          | message                | status | ts                |
+-----+-----+-----+-----+-----+
| 1  | mats@example.com | Adding employee mats    | OK     | 2010-01-13 15:50:45 |
| 2  | mats@example.com | Name change from ...    | OK     | 2010-01-13 15:50:48 |
| 3  | mats@example.com | Password change         | OK     | 2010-01-13 15:50:50 |
| 4  | mats@example.com | Removing employee       | OK     | 2010-01-13 15:50:52 |
| 5  | mats@example.com | Adding employee mats     | OK     | 2010-01-13 16:11:45 |
| 6  | mats@example.com | Adding employee mats     | FAIL   | 2010-01-13 16:12:00 |
+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)

```

看最后一行，状态是 `FAIL`。这一行是由 `before` 触发器的 `tr_employee_insert_before` 写入的。由于二进制日志严格记录了 `master` 上数据库的所有改变，倘若语句执行后为语句或触发器带来了一些非事务型变更，就需要把这个语句写入二进制日志。由于这个语句失败了，`after` 触发器的 `tr_employee_insert_after` 没有执行，所以 `before` 触发器执行以后状态就一直是 `FAIL`。

`master` 上语句执行失败，还要向二进制日志写入失败信息。MySQL 服务器通过向 `Query` 事件添加一个错误代码字段来实现，该字段记录导致语句失败的确切错误代码，随事件一起被写入二进制日志。

86 使用 `SHOW BINLOG EVENTS` 命令看不到这个错误代码，但是可以通过 `mysqlbinlog` 工具来查看，这点在本章后面会进行介绍。

## 将事务写入日志

截至目前，我们已经介绍了如何将单个语句写入二进制日志，包括上下文信息，但还没涉及事务。这一节我们将简单介绍如何将事务写入日志。

以下几种情况会开始一个事务：

- 当用户发出 `START TRANSACTION` 或 `BEGIN` 命令时。
- 当 `AUTOCOMMIT=1`, 且开始执行访问事务型表的语句时。注意, 只写非事务型表 (如 `MyISAM` 表) 的语句不会启动事务。
- 当 `AUTOCOMMIT=0`, 且上一个事务已经隐式 (执行的语句本身做了隐式提交) 或显式 (使用 `COMMIT` 或 `ROLLBACK`) 地被提交或终止时。

并不是事务开始后的每一个语句都属于这个事务。在处理二进制日志的时候需要特别考虑一些例外情况。

从定义上说, 非事务型语句不是事务的一部分。非事务型语句的执行会立即生效, 而不用等事务来提交。这也意味着它们无法回滚。它们不会影响活动事务: 任何在非事务型语句之后执行的事务型语句仍然属于当前活动的事务。

此外, 有些语句会做隐式提交。根据隐式提交的原因不同, 可以分成三组:

#### 写文件的语句

对大多数 DDL 语句 (如 `CREATE`、`ALTER` 等, 也有一些例外) 来说, 执行它们之前会对任何未完成的事务做一次隐式提交, 执行结束之后再做一次隐式提交。这些语句会修改文件系统中的文件, 所以不是事务型的。

#### 修改 `mysql` 数据表的语句

所有创建、删除或修改用户账户或用户权限的语句都是隐式提交的, 而且不是事务的一部分。这些语句修改了 `mysql` 数据库内部的表, 是非事务型语句。

在 `MySQL 5.1.3` 之前的版本中, 这些语句不会导致隐式提交, 但由于它们写的是非事务型表, 所以它们被当作非事务型语句。马上你将看到, 这会导致不一致, 所以经过若干版本后向这些语句添加了隐式提交。

87

#### 出于实践原因要求隐式提交的语句

很多情况下, 锁定表的语句、用于管理的语句和 `LOAD DATA INFILE` 语句都会导致隐式提交, 这是具体实现的需要。

导致隐式提交的语句明显不属于任何事务, 因为它们一旦执行任何活动事务都会被提交。想了解所有能够引起隐式提交的语句, 参见 `MySQL 参考手册`。

## 事务缓存

二进制日志中语句的顺序可能与它们实际的执行顺序不同, 因为它综合记录了每个事务的所有语句。服务器上的多个会话可能同时执行多个事务, 每个事务型存储引擎维护各自的事务日志以保证事务的正确执行。这些日志对用户来说是不可见的。与之不同的是,

二进制日志包含所有会话的所有事务信息，按照它们提交的顺序保存，就好像它们都是顺序执行的。

为了保证以事务为单位写二进制日志，服务器需要把语句分到不同的线程执行。提交事务的时候，服务器把所有这些语句一次性全部写入二进制日志。为此，服务器为每个线程保留了一个事务缓存（transaction cache），如图 4-4 所示。将语句存放在事务缓存中，当事务提交时，将事务缓存的内容复制到二进制日志，然后清空。

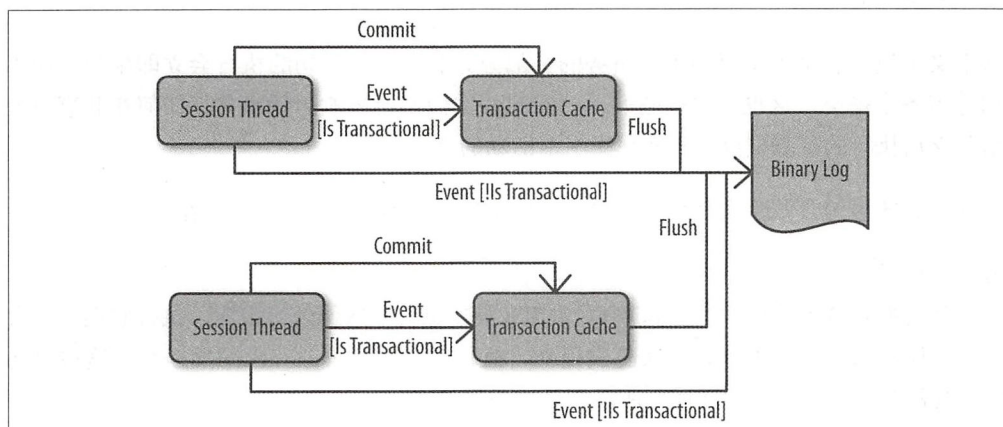


图4-4：线程、事务缓存和二进制日志

88 需要格外注意包含非事务型变更的语句。回顾一下，我们前面讨论过，非事务型语句不会导致当前事务的结束，因此执行非事务型语句带来的更新必须记录在某个地方且不会关闭当前活动的事务。如果这个语句同时影响事务型表和非事务型表，情况更加复杂。这些语句被看作是事务型的，但同时又包含了一些不属于事务的更新。

这时基于语句的复制无法正确处理所有可能的情况，因此需要一个更有效的方法。下面我们将介绍服务器采取的办法，然后是复制过程中需要注意的问题。

## 如何写入非事务型语句

如果没有活动事务，非事务型语句执行结束直接被写入二进制日志，无须通过事务缓存“中转”。但是，如果有活动事务，那么语句的处理规则如下：

1. 如果语句被标记为事务型的，则其被写入事务缓存。
2. 如果语句未被标记为事务型的，且事务缓存中没有语句，则将该语句直接写入二进制日志。
3. 如果语句未被标记为事务型的，但是事务缓存中有语句，则将该语句写入事务缓存。

规则可能看起来很奇怪，不过示例 4-15 可以帮助你理解其原因。还是用前面的 employee 表和 log 表，在示例 4-15 所示的事务中，我们先后更改了事务型表和非事务型表。

示例4-15：带有非事务型语句的事务

```
1  START TRANSACTION;
2  SET @pass = PASSWORD('xyzy');
3  INSERT INTO employee(name,email,password)
   VALUES ('mats','mats@example.com', @pass);
4  INSERT INTO log(email, message)
   VALUES ('root@example.com', 'This employee was bad');
5  COMMIT;
```

根据规则 3，第 4 行的语句将会被写入事务缓存，即使这个表是非事务型的。假定这个语句直接被写入二进制日志，它可能会跑到第 3 行语句前面，因为直到第 5 行提交成功后，第 3 行语句才会被写入二进制日志。简而言之，在 slave 日志中，第 4 行的 DBA 注释可能出现在第 3 行员工表的实际变更之前，这显然与 master 不一致。规则 3 避免了这样的情况。图 4-5 的左边部分所示的是没有规则 3 的错误结果，而右边部分是应用了规则 3 的实际情况。

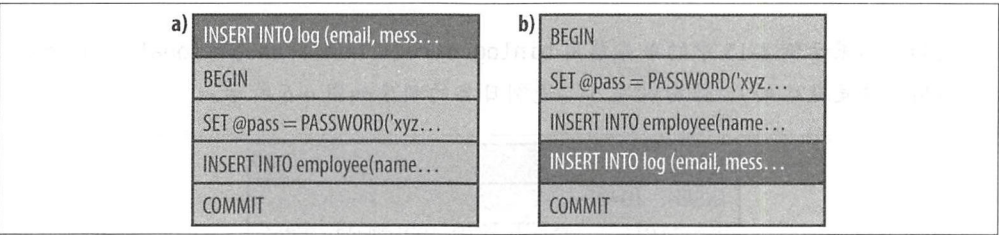


图4-5：是否应用规则3时的二进制日志

规则 3 其实是一个权衡问题。因为当执行事务的时候缓存非事务型语句，这存在这样的风险：两个事务在 master 上更新同一个非事务型表的顺序与它们写入二进制日志的顺序不同。

如果某个事务中第一个事务型语句和第二个非事务型语句之间有依赖关系，就会出现这种情况。但通常服务器无法处理这种情况，因为这要求完全解析每个语句（包括所有触发器调用的代码），并进行相关性分析。尽管技术上是可行的，但这样会给活动事务中的所有语句带来额外处理工作，所以会影响性能，可能影响还很大。不过问题几乎总是可以通过正确设计事务和确保事务中没有这种依赖关系来解决，所以并没有给 MySQL 带来开销。



如何避免非事务型语句的复制问题

要想没有问题最好不用非事务型表。但是，如果应用程序确实需要用到非事务型表，要避免前文所说的依赖关系，策略是：确保事务中影响非事务型表的语句先写入日志。这样的话，语句就会直接被写入二进制日志，因为事务缓存是空的（参考前一节的规则 2）。这些语句之间没有依赖关系。

如果事务后面的操作还需要从这些语句获取值，可以将它们赋值到临时表或变量。接下来就可以引用这些临时表或变量来执行事务了。

90

直接写入非事务型语句

从 MySQL 5.6 开始，可以通过 `binlog_direct_non_transactional_updates` 选项强制非事务型语句直接写入二进制日志。如果启用了这个选项，所有非事务型语句会在事务之前被写入二进制日志，这些语句只要出现在事务中而不必属于这个事务，甚至包括出现在事务型语句之后的非事务型语句。这就改变了基于语句的复制的特有行为。在基于行的复制中，非事务型语句的行总是写在事务之前的。因为基于行的复制不会执行任何语句而仅仅更改表中的数据，所以这样是可行的。因此，非事务型“语句”根本不存在依赖性问题的，可以安全地写在事务之前。

例如，如果示例 4-15 中的事务禁用 `binlog_direct_non_transactional_updates` 选项（这是默认的），语句被写入二进制日志的顺序如图 4-6 所示。

BEGIN
INSERT_ID=2
use 'test'; INSERT INTO log(email, mess
@'pass'=_utf8 0x2A31353141463642...
use 'test'; INSERT INTO employee(name
COMMIT

图4-6：示例4-15中语句的默认写入顺序

但是如果启用了 `binlog_direct_non_transactional_updates` 选项，事件顺序如图 4-7 所示。我们看到写 `log` 表是一个单独的事务，它在包含事务型语句的事务之前写入日志。



```
BEGIN
INSERT_ID=2
use 'test'; INSERT INTO log(email, mess
COMMIT

BEGIN
@'pass'=_latin1 0x2A31353141463642
use 'test'; INSERT INTO employee (name
COMMIT
```

图4-7: 示例4-15中语句的安全写入顺序

由于非事务型语句在事务之前写入，如果事务的某些语句在非事务型语句之间执行，那么要保证这些语句之间没有依赖关系。否则，就要考虑使用基于行的复制。

91

## 使用 XA 进行分布式事务处理

MySQL 5.0 版本可以通过 X/Open DTP（分布式事务处理）模型即 XA，协调处理涉及不同资源的事务。尽管 XA 目前没有被广泛使用，但它是协调各种事务资源的有力工具。

在版本 5.0 中，服务器内部使用 XA 来协调二进制日志和存储引擎。

还有一些允许客户端 XA 同步的命令。XA 可以将不同用户的不同语句当作单个事务来处理。但另一方面，它占用了一些开销，因此有些管理员会全程关闭它。

关于如何使用 XA 协议的内容超出了本书的讨论范围，不过我们将简单介绍一下 XA，以及它对二进制日志的影响。

XA 由一个事务管理器（transaction manager）和一组资源管理器（resource manager）组成，资源管理器将全局事务以原子单位提交，事务管理器则负责协调这些资源管理器。每个事务有一个唯一的 XID，供事务管理器和资源管理器使用。在 MySQL 服务器内部，事务管理器通常由二进制日志使用，而资源管理器由存储引擎使用。图 4-8 展示了提交 XA 事务的过程，它由两个阶段组成。

第 1 阶段，每个存储引擎都需要为提交做准备。在准备过程中，存储引擎将正确提交所需的一切信息写入安全存储，然后返回一个 OK 消息。只要有一个存储引擎返回的消息是否定的，就无法提交这个事务，提交被终止，所有引擎执行事务回滚。

当所有的存储引擎准备就绪且没有任何错误，尚未进入第 2 阶段之前，事务缓存被写入二进制日志。普通事务以带有 COMMIT 的普通 Query 事件结束，而 XA 事务以一个包含 XID 的 Xid 事件结束。

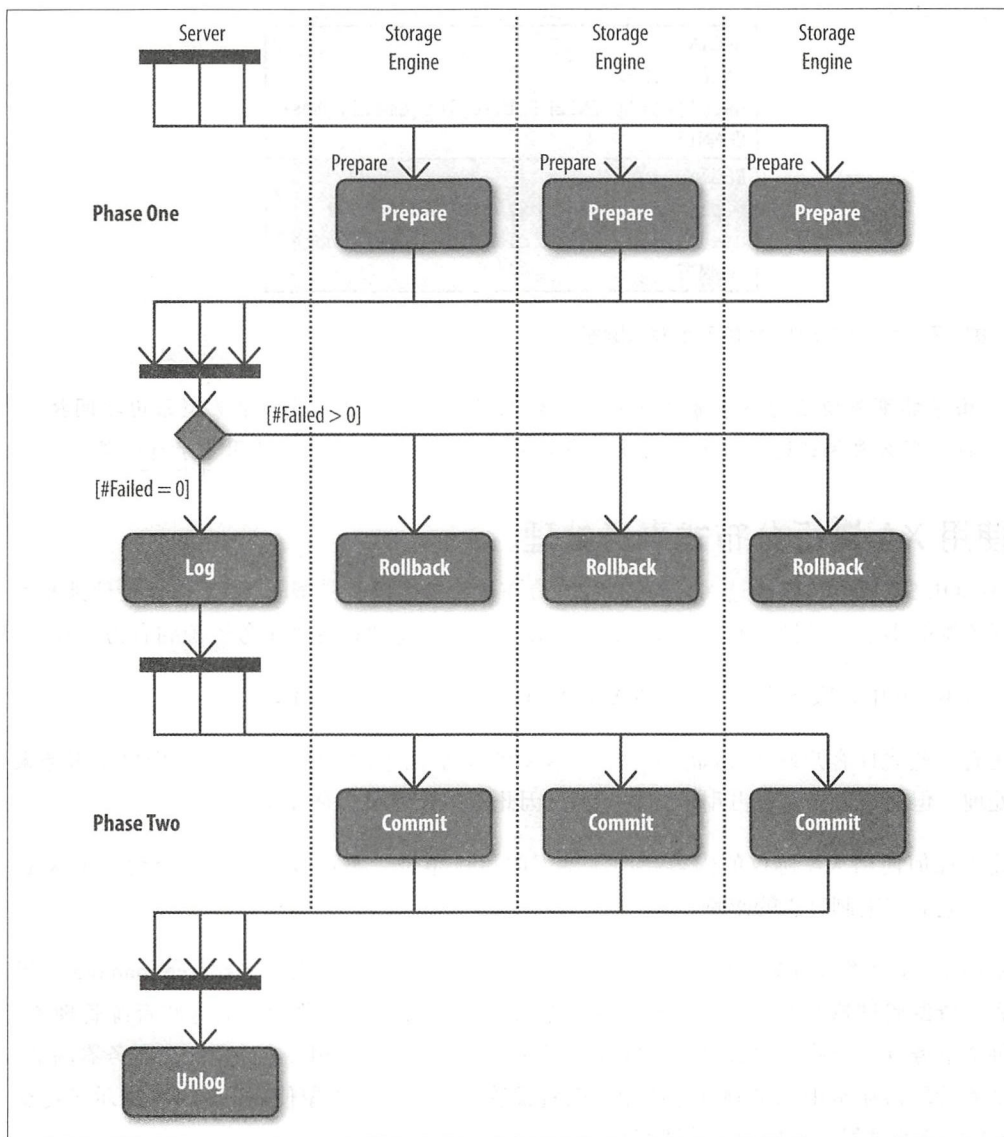


图4-8：使用XA提交分布式事务

- 92 第2阶段，阶段1准备就绪的所有存储引擎要提交事务。在提交时，每个存储引擎都要确认在稳定存储中提交了事务。注意，理解提交不可能失败：一旦通过了阶段1，存储引擎能够保证事务的提交，因此阶段2不会报错。当然硬件错误会导致崩溃，不过鉴于
- 93 存储引擎将信息保存在持久性存储器中，所以能够在服务器重启后正确恢复数据。重启过程参见本章后面“二进制日志和系统崩溃安全”一节。

阶段 2 完成之后，事务管理器就可以丢弃共享资源，而且应该要丢弃它们。由于二进制日志不需要做这样的清理工作，所以在这一步 XA 什么也不用做。

如果提交 XA 事务的时候发生系统崩溃，服务器重启后进入恢复过程，如图 4-9 所示。启动后，服务器打开上一个二进制日志文件并检查 `Format_description` 事件。如果 `binlog-in-use` 标记被设置（参见本章后面“Binlog 文件轮换”一节），说明服务器发生了崩溃，需要执行 XA 恢复。

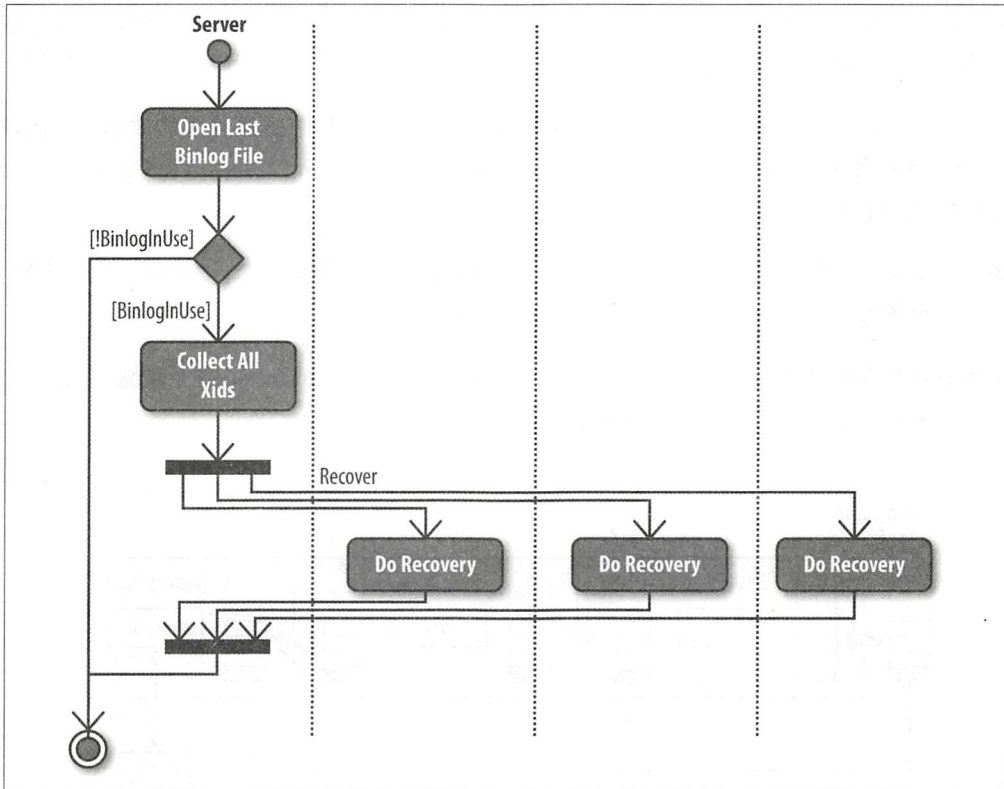


图4-9: XA恢复的过程

首先服务器检查刚刚打开的二进制日志，通过读取 Xid 事件确定所有事务的 XID。然后服务器中加载的存储引擎将根据这个清单来提交事务。对于每个 XID，存储引擎将首先确定该 XID 对应的事务是否准备就绪，如果还没提交则提交这个事务。如果存储引擎准备好的事务 XID 不在这个清单中，那么显然这个 XID 在服务器崩溃前没有写入二进制日志，因此这个事务需要回滚。

## 二进制日志的组提交

实际上将数据写到磁盘非常费时间。磁盘性能比内存慢好几个数量级，磁盘访问时间是按毫秒计算的，而内存则以纳秒计算。出于这个原因，操作系统拥有复杂的内存管理系统，将一部分文件放在内存，除非必要否则不用再做写磁盘操作。由于数据库系统必须能够安全应对崩溃情况，所以需要在事务提交的时候，强制将数据写回磁盘。

如果每个事务都要写磁盘会带来性能问题，为了避免这个问题，多个独立事务可以按组的形式，一起写入磁盘，这就是组提交（group commit）。由于写磁盘的时间取决于磁盘头移动到正确磁盘位置的时间，与写入多少数据量无关，所以这样能够极大地提高性能。

但是，不仅要考虑存储引擎提交事务数据的效率，还要考虑写二进制日志的效率。为此，MySQL 5.6 增加了二进制日志组提交（binary log group commit）功能，即多个独立事务可以按组的形式提交到二进制日志，从而大大提高了性能。

为了保证在线备份工具（如 MySQL Enterprise Backup）的正确操作，要确保写入二进制日志的事务顺序与它们在存储引擎中的准备顺序相同。

实现方式是在每个事务完全提交之前增加一些步骤，如图 4-10 所示。每个步骤引入了一个互斥对象（mutex），确保每个步骤最多只有一个线程。

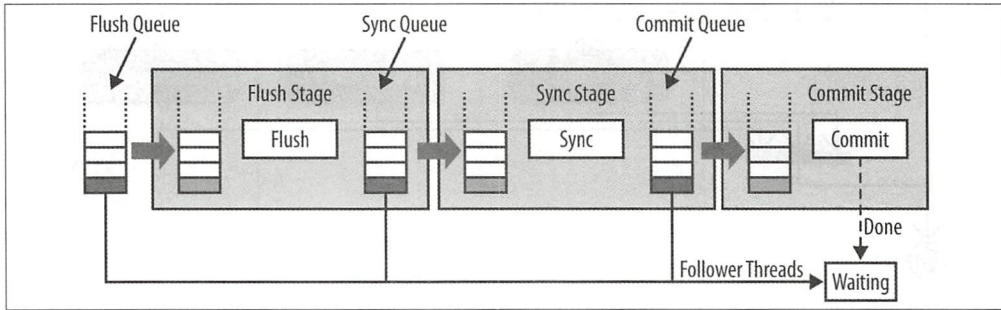


图4-10: 二进制日志的组提交架构

各个步骤负责处理一部分提交过程。第一步将线程的事务缓存到文件页，第二步执行一个同步操作将文件页写到磁盘，最后一步提交所有事务。

95 为了保证各个步骤之间会话的有序性，每个步骤都有一个队列，保存一系列等待处理的会话。每个队列由一个互斥对象提供保护，即操作队列的时候将持有这个互斥对象。表 4-1 列出了这些互斥对象的名字，可以用这些结构监控提交阶段的性能。



表4-1：二进制日志提交的三个步骤及其互斥对象

步骤	步骤互斥体	步骤队列互斥体
flush	LOCK_log	LOCK_flush_queue
sync	LOCK_sync	LOCK_sync_queue
commit	LOCK_commit	LOCK_commit_queue

通常一个会话对应一个线程，在 flush 步骤，任何想要提交事务的会话线程会进入队列。会话线程的处理过程如下。

1. 如果会话线程进入一个非空队列（又称 follower），需要等待队列中的其他会话线程提交事务。
2. 如果会话线程进入一个空队列（又称 leader），负责登记所有进入该队列的会话。
3. leader 有序地一次性清空队列中的所有会话，然后新的会话可以再次进入队列。
4. 每个步骤的处理如下：
  - flush 步骤，每个会话的事务按照它们进入 flush 队列的顺序被 flush 到二进制日志。
  - sync 步骤，执行一个 fsync 调用。
  - commit 步骤，存储引擎中事务的提交顺序与它们登记的顺序一致。

96

5. 会话进入下一个步骤的队列的入队顺序与它们在当前步骤的登记顺序一致。  
注意，此时队列可能是非空的，说明会话线程正好“赶上”了上一个 leader 线程。这个时候原本应该是 leader 的会话线程就成了下一步骤的 follower，而已经在队首的 leader 负责处理线程集合。注意，leader 可能变成 follower，而 follower 不可能成为 leader。

新的 leader 线程会将提交过程进行到一半的旧线程合并到会话队列中，从而系统可以动态适应各种情况。通常，sync 步骤是最慢的，所以很多线程完成 flush 步骤之后会“堆”在这一步，而只有一个 leader 线程及其带领的会话被处理。如果使用电池后备高速缓存，fsync 调用就很快，会话可能就不会堆在这一步了。

事务必须按顺序提交，这样在线备份方法（如 XtraBackup 或 MySQL Enterprise Backup）才能正确操作。为此，commit 步骤按照事务写入二进制日志的顺序提交它们。

但是，commit 步骤可能出现平行提交事务的情况，即提交顺序是任意的。这并不会影响普通操作的正确性，但是做平行提交的时候不应该执行在线备份。

基准测试说明，平行提交事务并不会明显提高性能，所以默认情况下总是按顺序提交。除非有特殊需要（或是为了测试），建议按顺序提交。

通过 binlog\_order\_commits 控制事务是否按序提交。如果设为 OFF，则平行提交事务，而且没有最后一个步骤（线程本身以任意顺序提交，而不用等待 leader）。

当从队列读取会话的时候，flush 步骤做了优化。进入 flush 步骤并不是处理整个队列，leader 每次从队列中“搬”一个会话，flush 它，然后重复这个过程。优化的思想是：只要还有会话入队，就没必要处理整个队列，而且将事务 flush 到二进制日志很费时间，按会话逐个处理能够让更多需要提交的会话入队。该优化使得吞吐效率极大提升。但这会产生延迟，可以通过 `binlog_max_flush_queue_time` 参数控制 leader 线程从 flush 队列“搬”会话的时间，单位为毫秒。一旦超时，取回整个队列，leader 重新恢复该步骤的处理，如前面讲过的那样。也就是说，这个时间值并不是事务的延迟阈值：leader 必须在进入 sync 步骤之前 flush 掉队列中登记过的所有事务。

97

## 基于行的复制

复制的主要目的是同步 master 和 slave，保证它们拥有相同的数据。前面已经知道，复制提供了很多特性来保证 master 和 slave 上的结果尽可能一致，比如上下文事件、会话特定的 ID 等。

尽管如此，有时候，基于语句的复制仍然无法正确处理：

- 如果 UPDATE、DELETE 或 INSERT 语句包含 LIMIT 从句，那么语句执行的时候倘若数据库崩溃，则可能导致错误。
- 如果非事务型语句执行期间出错，无法保证 master 和 slave 上的数据变化一致。
- 如果语句含有 UDF 函数调用，则无法保证 slave 上的值与 master 上的相同。
- 如果语句含有任何不确定的函数，例如 USER、CURRENT\_USER、CONNECTION\_ID 等，可能会导致 master 和 slave 上的结果不一致。
- 如果语句同时更新两个含有 autoincrement 字段的表，则无法保证正确性。因为只有最后插入的那个 ID 会被复制到 slave 上然后同时应用到两张表中去，而对 master 来说，每张表的插入 ID 是不同的。

上述情况下，最好复制那些插入表中的真实数据，这就是基于行的复制（row-based replication）。

基于行的复制不复制产生变更的语句，而是复制每个被插入、删除或更新的行。发给 slave 的行与发给存储引擎的行是一样的，其中包含插入表的真实数据。所以不用考虑 UDF，不用跟踪 autoincrement 计数器，也不用考虑语句的部分执行——只需简单地考虑数据本身即可。

98

基于行的复制解决了基于语句的复制无法完成的事情。但要注意它们之间的区别。

选择基于语句的复制还是基于行的复制，需要考虑以下问题：

- 这些语句是否会更新大量的行，还是只做少量行的更新或插入？

如果语句要更新大量的行，那么基于语句的复制更快，因为语句相比行少得多。但是由于语句也要在 slave 上执行，所以事实并不总是这样。如果语句的优化和执行计划很复杂，这时可能基于行的复制更快，因为寻找行的逻辑快得多。

如果语句只更新或插入少量行，则基于行的复制更快，因为不需要解析直接交给存储引擎处理。

- 是否需要知道执行了哪些语句？至少可以说，在基于行的复制中，事件难以解码；而在基于语句的复制中，语句被写入二进制日志中，因此可以直接读取。<sup>注 2</sup>
- 基于语句的复制的复制模型很简单：只要在 slave 上执行相同的语句即可。这种技术应用已久，很多 DBA 对其较熟悉，而不太熟悉基于行的复制。所以如果基于行的复制出现故障，可能比较难解决。
- 如果 master 和 slave 上的数据不同，执行语句的结果也会不同。有时是故意的（这时应该使用基于语句的复制），但有时是无意的，这就需要通过基于行的复制来避免这种情况。

## 启用基于行的复制

写二进制日志的时候通过 `binlog-format` 选项可以控制使用哪种格式。该参数值既可以作为全局变量也可以作为会话变量，取值为 `STATEMENT`、`MIXED` 或者 `ROW`（参见本章后面的“基于行的复制参数”一节）。从而会话可以暂时切换复制格式（但更改这个参数需要 `SUPER` 权限，哪怕是会话变量）。但是，为了确保服务器启动时使用的是基于行的复制，需要停止 master 并更改配置文件。示例 4-16 给出了启用基于行的复制需要添加的设置。

99

示例4-16：配置基于行的复制参数

```
[mysqld]
user          = mysql
pid-file      = /var/run/mysqld/mysqld.pid
socket        = /var/run/mysqld/mysqld.sock
port          = 3306
basedir       = /usr
datadir       = /var/lib/mysql
tmpdir        = /tmp
log-bin       = master-bin
log-bin-index = master-bin.index
server-id     = 1
binlog-format = ROW
```

注 2 MySQL 5.6 支持将产生行的语句连同行一起写入日志。参见本章后面的“基于行的复制参数”一节。



## 使用混合模式

MySQL 5.1 及其以后的版本推荐使用混合模式的复制，但 `binlog-format` 参数的默认值为 `STATEMENT`。看上去很奇怪，但这样却避免了从 5.0 或早期版本升级带来的问题。因为旧版本没有基于行的复制，只能使用基于语句的复制，而且 MySQL 开发者也不想突然切换复制方式。如果服务器升级时突然开始发送基于行的复制事件，那么部署将变得一团糟。为了让 DBA 在升级的时候少操点心，将这个参数的默认值还设为 `STATEMENT`。

但如果你用的是 MySQL 5.1 版本，就会发现 `binlog-format` 参数的值为 `MIXED`，这也是推荐的值。

混合模式复制的原理很简单：正常情况下使用基于语句的复制，而对不安全的语句则切换到基于行的复制。我们前面已经解释过哪些语句可能会带来问题及其原因。总结一下，当出现以下情况时，混合模式需要切换到基于行的复制：

- 该语句调用了：
  - `UUID` 函数；
  - 用户自定义函数；
  - `CURRENT_USER` 或 `USER` 函数；
  - `LOAD_FILE` 函数。
- 一个语句同时更新了两个或两个以上含有 `AUTO_INCREMENT` 列的表。
- 语句中使用了服务器变量。
- 存储引擎不允许使用基于语句的复制，例如，MySQL Cluster 引擎。

当然，以上所列的情况并不完整：随着新的不安全因素被发现，这个列表也在不断扩展。更加完整精确的列表参见在线 MySQL 参考手册。

## 二进制日志管理

目前我们提到的事件代表了 `master` 上的数据的实际变更，但某些不代表数据变更的事情也会影响复制。例如，如果服务器停止了，也会潜在地影响复制。当然了，服务器不运行的时候不会有任何事件写入二进制日志，但这意味着数据文件的任何更新都不会反映在二进制日志中。一个典型的例子是恢复备份及操作数据文件。因为服务器不运行，所以这样的变更不会被复制。有时候服务器停止这件事会在二进制日志中以一个二进制事件精确表示，但二进制日志中的记录与事情的真实发生之间可能是有（时间）差距的。

事件也可以用于其他目的。因为二进制日志由多个文件组成，有必要将它们分割成适当的组，构成一个 `binlog` 文件序列。为了操作安全，向二进制日志添加一些特殊事件（轮换事件）。



## 二进制日志和系统崩溃安全

我们已经知道，写二进制日志与 master 数据库中的数据变化并不是一一对应的关系。要保持数据库和二进制日志之间的一致性，以防系统崩溃，这点很重要。换句话说，如果更新没有写入二进制日志，就不会被提交到存储引擎，反之亦然。

非事务型引擎马上就有问题了。例如，无法保证二进制日志和 MyISAM 表之间的一致性，因为 MyISAM 是非事务型的，存储引擎在语句写入日志之前，早就完成了所有变更操作。

但是对于事务型存储引擎来说，MySQL 有一系列措施确保系统崩溃并不会导致二进制日志丢失过多的信息。

正如我们在本章前面“将语句写入日志”一节所描述的那样，事件写入二进制日志的时机是表上的锁释放之前、所有变更交给存储引擎之后。因此如果在存储引擎释放锁之前系统崩溃，服务器必须确保写入二进制日志的变更实际存在于磁盘上的表中，然后才能提交语句（或事务）。这需要与标准文件系统之间的同步与协调。

101

由于磁盘访问比内存访问开销大得多，操作系统将文件的一部分缓存在内存的某个特殊位置——通常称之为页面缓存（page cache），等到必要的时候才将文件数据写入磁盘。当不得不从磁盘加载另一个页面且页面缓存已满，就需要写磁盘了。当然，应用程序也可以通过一个显式调用要求将页面写入磁盘。

回顾前面讨论过的 XA，当第一个阶段完成后，所有数据被写入持久性存储器（即磁盘），以正确应对崩溃。也就是说，每次提交事务时，页面缓存都会被写入磁盘。这样很费时，而且根据不同的应用程序，也并不总是必要的。通过设置 `sync-binlog` 选项可控制数据写入磁盘的频率。其值是一个整数，指定二进制日志写入磁盘的频率。例如，如果将这个选项设置为 5，即每提交语句或事务 5 次，二进制日志写入磁盘 1 次。默认值是 0，即服务器不会显式将二进制日志写入磁盘，而是由操作系统决定。



注意在 MySQL 5.6 中，由于引入了二进制日志的组提交，所以 `sync` 以提交组为单位，而不是按事务或语句。也就是说，`sync-binlog=1` 其实是若干事务批量写入磁盘。更多内容请阅读本章前面“二进制日志的组提交”一节。

对于支持 XA 的存储引擎，例如 InnoDB，设置 `sync-binlog=1`，一般的系统崩溃情况下，都不会丢失任何事务。而不支持 XA 的引擎，可能会至少丢失一个事务。

但是，如果每个组都写一次磁盘，性能会受影响，通常还很糟糕。磁盘访问相当慢，所以用缓存来提高性能，不需要总是将数据写入磁盘。如果你愿意冒险丢失几个事务或语句，比如你可以通过手动恢复来处理这些损失，或者应用程序根本不在乎这几个事务或

语句，那么你可以将 `sync-binlog` 的值设置得高一点，否则直接使用默认值。

## binlog 文件轮换

MySQL 启动一个新文件来定期保存二进制日志事件。由于实践和管理的原因，一直向单个文件写日志是不可行的——操作系统对文件大小有限制。前面提到过，服务器当前正在写入的文件称为活动（active）binlog 文件。

根据上下文不同，切换到新文件的过程称为二进制日志轮换，或者 binlog 文件轮换。

主要有 4 种行为会导致轮换：

### 服务器停止

每次服务器启动，都会开始一个新的二进制日志。稍后我们将讨论为什么。

### binlog 文件大小达到最大限制

如果 binlog 文件太大，它将自动轮换。可以通过服务器变量 `binlog-cache-size` 来控制 binlog 文件的大小。

### 二进制日志被显式刷新

`FLUSH LOGS` 命令将所有日志写入磁盘，然后创建一个新文件继续写二进制日志。当 PITR 管理恢复镜像时，这是很有用的。从一个打开的 binlog 文件可能会读到预料之外的结果，因此建议在试图使用 binlog 文件执行恢复前，强制进行显式刷新（flush）。

### 服务器上发生事故

除了完全停止工作以外，服务器还可能遇到其他事故，从而导致二进制日志被轮换。这些事故有时候要求管理员进行特殊的人工干预，因为它们可能会在复制流中留下“缺口”。如果在事故后服务器以一个新的 binlog 文件开始，DBA 处理这个事故会更加容易一些。

binlog 文件的第一个事件是 `Format_description` 事件，描述了写 binlog 文件服务器以及该文件的内容和状态信息。

有三个特别有趣的地方：

### binlog-in-use 标记

因为系统崩溃可能发生在服务器正在写 binlog 文件的时候，所以一定要在文件正确关闭的时候加个标记。否则，DBA 在 master 或 slave 上重放的可能是已损坏的文件，然后会导致更多问题。为了保证文件的完整性，在创建文件，以及写入最后一个事件（Rotate）后清空该文件的时候，设置 `binlog-in-use` 标记。这样，任何程序都

可以看到 binlog 文件是否已经被正确关闭。

## binlog 文件格式版本

在 MySQL 的发展历程中，二进制日志文件的格式经历了多次变化，而且以后一定还会再变。每次发生重大改变（通用头的显著改变）时，开发者都会增加格式的版本号，增加一些旧版本服务器不可读的文件。（从 MySQL 5.0 版本开始，当前格式是版本 4。）binlog 文件格式的版本字段列出了它的版本号，如果某个不同版本的服务器不能处理这个版本的文件，就直接拒绝读取该文件。

103

## 服务器版本

这是一个字符串，表示写该日志文件的服务器的版本。比如，本章的例子使用的服务器版本是“5.5.31-0ubuntu0.12.04.1”。你会发现，这个字符串一定包含 MySQL 服务器的版本，同时还包含 build 相关的额外信息。有些情况下，当复制发生在不同版本的服务器之间时，这些信息可以帮助开发者找出并解决其中的微小错误。

为了在系统崩溃的时候也能安全地轮换二进制日志，服务器采用预写（write-ahead）策略，并把这个意图存到一个名为清除索引文件（purge index file）的临时文件中（之所以叫这个名字，是因为等会儿你会发现，这个文件还会用于清除 binlog 文件）。它的名字是基于索引文件的，例如，如果索引文件的名称是 *master-bin.index*，清除索引文件的名称就叫 *master-bin.~rec~*。创建新的 binlog 文件，并更新索引文件指向它之后，服务器就会删除清除索引文件。



在 MySQL 5.1.43 之前的版本中，轮换或清除 binlog 文件可能会残留孤立文件，即文件已经不在索引文件里但仍存在于文件系统中。所以，旧文件可能没有正确清除导致有残留，需要人工清理文件。

孤立文件不会给复制带来问题，但却是个麻烦。本节讲述的是没有孤立文件的情况下发生系统崩溃的情况。

如果发生系统崩溃，并且服务器上有清除索引文件，那么服务器将在启动的时候比较清除索引文件和索引文件，确定哪些是意图要做的，哪些是已经完成的。

5.6 版本之前，二进制日志可能被部分写入，比如在服务器崩溃前只有部分缓存被写入二进制日志时。在 MySQL 5.6 版本中，二进制日志被调整为删除那些部分写入的事务。这样做是安全的，因为只有完全写入二进制日志中的事务才会在存储引擎中提交。



## 事故

术语“事故”(incident)是指在服务器上不会产生数据变更但是必须写入二进制日志的事件,因为它们潜在地影响了复制。大多数事故不需要 DBA 的特殊干预,例如,停止服务器然后重启,但不改变数据库文件,这个过程中不可避免地产生一些需要特殊处理的事故。

现在,你会发现,二进制日志中有以下两种 incident 事件。

### Stop

表示服务器正常停止。如果服务器发生系统崩溃,即使服务器重启,也不会写入 Stop 事件。该事件被写入旧的 binlog 文件(重启服务器会轮换到新文件),而且事件中仅包含一个通用头,没有其他信息。

在 slave 上重放二进制日志时,将忽略任何 Stop 事件。通常,服务器曾经停止运行这件事不需要特别留意,复制可以照常进行。如果服务器停止的时候切换到了新版本,下一个 binlog 文件中就会有标记,如果服务器读取二进制日志的时候无法处理新版本的 binlog 文件格式,服务器就会停止。从这个意义上说,Stop 事件无法代表复制流中的“缺口”。然而,这个事件还是值得记录的,因为可能有人在重启复制前手动地恢复了一个备份或者对文件做了其他变更,这样 DBA 在重放文件时就会看到这个事件,以便在适当的时候启动或停止重放。

### Incident

这是版本 5.1 中引入的事件类型,表示通用的事故(incident)事件。与 Stop 事件不同,该事件包含一个标识符指定发生的事故类型。该事件表明服务器被迫执行了一些操作,几乎导致二进制日志变更丢失。

例如,如果数据库重新装载,或者非事务型事件由于过大而无法写入 binlog 文件,版本 5.1 就会写入 Incident 事件。当 MySQL 集群中的某个节点重新装载数据库从而导致可能不同步时,就会产生这个事件。

当 slave 重放二进制日志时,如果遇到 Incident 事件,就会因错误而停止。如果在 MySQL 集群重载时发现 Incident 事件,表明需要重新同步集群,很可能还要找到那些 binlog 文件中丢失的事件。

## 清除 binlog 文件

随着时间的推移,除非清除文件系统中的旧文件,否则服务器上的 binlog 文件会越来越多。服务器可以自动清除文件系统中旧的二进制日志,或者也可以明确告诉服务器清除文件。



设置 `expire-logs-days` 选项，服务器可以自动清除旧的 binlog 文件。这个选项也可用作服务器变量，其值为存储 binlog 文件的天数。记住，同所有的服务器变量一样，该设置重启后不保存。因此，如果希望重启后仍保持自动清除设置，必须把这个设置添加到服务器的 `my.cnf` 文件中。

要想手工清除 binlog 文件，可使用 `PURGE BINARY LOGS` 命令，有以下两种形式。

#### `PURGE BINARY LOGS BEFORE datetime`

这个命令清除给定时间前的所有文件。如果 `datetime` 在某个日志文件的中间（通常都是这样），那么 `datetime` 所在文件之前的所有文件都将被清除。

#### `PURGE BINARY LOGS TO 'filename'`

这个命令清除给定文件前的所有文件。即，在 `SHOW MASTER LOGS` 命令显示的所有文件中，`filename` 前的文件都将被删除，从而 `filename` 变成第一个 binlog 文件。

当服务器启动或二进制日志轮换完成时，binlog 文件将被清除。如果服务器发现文件需要被清除，要么因为文件是比 `expire-logs-days` 旧，要么因为执行了 `PURGE BINARY LOGS` 命令。清除文件的过程是，首先将那些服务器认为时机成熟的文件写到清除索引文件，然后，将文件从文件系统中删除，最后删除清除索引文件。

如果发生崩溃，服务器通过比较清除索引文件和索引文件的内容来继续删除文件，并删除那些因系统崩溃而没有被删除的文件。前面讲过，清除索引文件也会在轮换时使用，因此如果在索引文件正确更新之前发生崩溃，新的 binlog 文件将被删除，然后在每次轮换时重新创建 binlog 文件。

## mysqlbinlog 实用工具

对管理员十分有用的一个工具是客户端程序 `mysqlbinlog`。这是一个小程序，可以审查 binlog 文件及中继日志文件的内容（我们将在第 8 章讨论中继日志）。除了在本地读取 binlog 文件，`mysqlbinlog` 同样可以从其他服务器上远程读取 binlog 文件。

`mysqlbinlog` 不仅在审查复制中的问题时非常有用，它还可以用于实现 PITR，就像第 3 章中演示的那样。

一般的，`mysqlbinlog` 输出二进制日志的内容，该内容可以被发送到运行中的服务器上，且格式是可执行的。如果使用基于语句的复制，命令行中的执行语句就是 SQL 语句。如果使用基于行的复制，`mysqlbinlog` 需要一些额外数据来处理，这将在第 8 章介绍。本章只讨论基于语句的复制，因此这里使用的命令会带有一些选项，使我们不必处理基于行的复制。

本节会介绍 *mysqlbinlog* 的某些选项，如果需要了解全部，请参考在线 MySQL 参考手册。

## 基本用法

下面让我们从一个简单的例子开始。我们创建了一个 binlog 文件，然后用 *mysqlbinlog* 查看它。启动一个客户端连接到 master，然后执行以下命令查看它们在二进制日志中的样子：

```
mysql> RESET MASTER;
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE employee (
    ->   id INT AUTO_INCREMENT,
    ->   name CHAR(64) NOT NULL,
    ->   email CHAR(64),
    ->   password CHAR(64),
    ->   PRIMARY KEY (id)
    -> );
Query OK, 0 rows affected (0.00 sec)

mysql> SET @password = PASSWORD('xyzyz');
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO employee(name,email,password)
    ->   VALUES ('mats','mats@example.com',@password);
Query OK, 1 row affected (0.01 sec)
```

```
mysql> SHOW BINARY LOGS;
+-----+-----+
| Log_name          | File_size |
+-----+-----+
| mysql-bin.000038 | 670       |
+-----+-----+
1 row in set (0.00 sec)
```

现在我们使用 *mysqlbinlog* 工具转储 binlog 文件 *master-bin.000038* 的内容，所有命令都会存在这个文件里。示例 4-17 给出了输出结果，其中为了适合页面，编辑上有些许变动。

示例4-17：执行mysqlbinlog的输出

```
$ sudo mysqlbinlog \
> --short-form \
> --force-if-open \
```

```

> --base64-output=never \
> /var/lib/mysql1/mysqld1-bin.000038
1 /*!40019 SET @@session.max_insert_delayed_threads=0*/;
2 /*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
3 DELIMITER /*!*/;
4 ROLLBACK/*!*/;
5 use test/*!*/;
6 SET TIMESTAMP=1264227693/*!*/;
7 SET @@session.pseudo_thread_id=999999999/*!*/;
8 SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=1,
    @@session.unique_checks=1, @@session.autocommit=1/*!*/;
9 SET @@session.sql_mode=0/*!*/;
10 SET @@session.auto_increment_increment=1,
    @@session.auto_increment_offset=1/*!*/;
11 /*!\C utf8 *//*!*/;
12 SET @@session.character_set_client=8,@@session.collation_connection=8,
    @@session.collation_server=8/*!*/;
13 SET @@session.lc_time_names=0/*!*/;
14 SET @@session.collation_database=DEFAULT/*!*/;
15 CREATE TABLE employee (
16   id INT AUTO_INCREMENT,
17   name CHAR(64) NOT NULL,
18   email CHAR(64),
19   password CHAR(64),
20   PRIMARY KEY (id)
21 ) ENGINE=InnoDB
22 /*!*/;
23 SET TIMESTAMP=1264227693/*!*/;
24 BEGIN
25 /*!*/;
26 SET INSERT_ID=1/*!*/;
27 SET @`password`:=_utf8 0x2A31353141463... COLLATE `utf8_general_ci`/*!*/;
28 SET TIMESTAMP=1264227693/*!*/;
29 INSERT INTO employee(name,email,password)
30 VALUES ('mats','mats@example.com',@password)
31 /*!*/;
32 COMMIT/*!*/;
33 DELIMITER ;
34 # End of log file
35 ROLLBACK /* added by mysqlbinlog */;
36 /*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;

```

对输出结果，我们可以使用三个选项：

## short-form

该选项只输出发出的 SQL 语句信息，而忽略二进制日志中事件的注释信息。当仅使用 *mysqlbinlog* 将事件重放到服务器时，这个选项是非常有用的。如果想审查 binlog 文件中的错误，就需要注释信息，不能使用该选项。

## force-if-open

如果 binlog 没有被正确关闭，比如 binlog 文件仍在写入或服务器崩溃，*mysqlbinlog* 都将输出一条警告说这个 binlog 文件没有被正确关闭。该选项用于禁止输出那条警告。

## base64-output=never

该选项阻止 *mysqlbinlog* 输出 base64 编码的事件。如果要输出 base64 编码的事件，也会输出二进制日志的 Format\_description 事件，表明使用了哪种编码。这对基于语句的复制是不必要的，因此使用这个选项来阻止该事件。

在示例 4-17 中，1 ~ 4 行显示了每次输出前面的序。第 3 行设置了一个文件中不可能出现的分隔符。这个分隔符也是注释，处理语言不解释分隔符。

第 4 行的回滚目的是防止输出不小心包含了的事务，因为在输出发送到客户端之前，事务已经在客户端上开始了。

我们暂时跳到输出的末尾，即第 33 ~ 35 行，它们与 1 ~ 4 行相对应。它们恢复设置的值，并回滚所有活动的事务。这很有必要，比如在事务中间截断 binlog 文件的时候，这么做可以阻止该输出之后的任何 SQL 代码被包含到该事务中。

每当数据库变更，都会打印第 5 行的 USE 语句。在每个 SQL 语句前的二进制日志都指定了当前数据库，*mysqlbinlog* 只显示对当前数据库的变更，新事件的第 1 行才会出现 USE 语句。

每个事件的第 1 行都是 SET TIMESTAMP，如第 6 行和第 23 行。该语句提供了事件开始执行时的时间戳，是自 1970 年 1 月 1 日 (00:00:00 GMT) 以来的秒数。

第 7 ~ 14 行是一些常规设置，与第 5 行的 USE 类似，只有第一个事件或者参数值发生改变时，才会输出这些行。

第 29 ~ 30 行的 INSERT 语句使用用户定义变量向表中插入 autoincrement 类型的字段，在此之前设置了两个变量：第 26 行的会话变量 INSERT\_ID 和第 27 行的用户定义变量。这是因为二进制日志有 Intvar 和 User\_var 事件。

如果忽略 short-form 选项，在输出结果中，每个事件前面都会有些注释。从示例 4-18



中可以看到，注释以井号（#）开头。

示例4-18：解释mysqlbinlog输出中的注释

```
$ sudo mysqlbinlog \
> --force-if-open \
> --base64-output=never \
> /var/lib/mysql1/mysql1-bin.000038
.
.
.
1 # at 386
2 #100123 7:21:33 server id 1 end_log_pos 414 Intvar
3 SET INSERT_ID=1/*!*/;
4 # at 414
5 #100123 7:21:33 server id 1 end_log_pos 496 User_var
6 SET @`password`:=_utf8 0x2A313531...838 COLLATE `_utf8_general_ci`/*!*/;
7 # at 496
8 #100123 7:21:33 server id 1 end_log_pos 643
  Query thread_id=6 exec_time=0 error_code=0
9 SET TIMESTAMP=1264227693/*!*/;
10 INSERT INTO employee(name,email,password)
11 VALUES ('mats','mats@example.com',@password)
12 /*!*/;
13 # at 643
14 #100123 7:21:33 server id 1 end_log_pos 670 Xid = 218
15 COMMIT/*!*/;
16 DELIMITER ;
17 # End of log file
18 ROLLBACK /* added by mysqlbinlog */;
19 /*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

注释的第1行是事件的字节位置，第2行是该事件的其他信息。例如考虑 INSERT 语句行：

```
# at 496
#100123 7:21:33 server id 1 end_log_pos 643  Query  thread_id=6
      exec_time=0      error_code=0
```

这个注释的各部分的意义如下：

at 496

事件开始的字节位置，即事件的第1个字节。

100123 7:21:33

事件的时间戳，datetime 类型（日期加上时间）。这是开始执行查询的时间，或事件被写入二进制日志的时间。

server\_id 1

产生该事件的服务器 ID。这个服务器 ID 用来设置会话变量 `pseudo_thread_id`，如果这个事件是线程特定的且服务器 ID 与之前输出的 ID 不同，那么设置这个变量的行也会被输出。

110 end\_log\_pos 643

该事件后面的事件的字节位置。这个值与事件开始位置的差，就是该事件的长度。

Query

事件类型，在示例 4-18 中可以看到几种不同的事件类型，例如 `User_var`、`Intvar` 和 `Xid`。

接下来是事件特定（event-specific）的字段，即每个事件各不相同。对于 Query 事件，有这两个附加字段：

thread\_id=6

执行该事件的线程 ID，用于处理线程特定的查询，例如访问临时表的查询。

exec\_time=0

查询执行的时间，单位为秒。

示例 4-17 和示例 4-18 转储了单个文件的输出，但是 *mysqlbinlog* 也能接受多个文件。如果给定多个 binlog 文件，将会按顺序处理。

文件将按照请求顺序输出，而且并不检查文件末尾的 Rotate 事件是否指向下一个文件。确保这些 binlog 文件能够组成真正的二进制日志是用户的责任。

由于 binlog 文件的命名方式，向 *mysqlbinlog* 提交多个文件通常不是问题，例如通过使用 \* 作为文件名匹配的通配符。如果使用 binlog 文件计数器作为文件扩展名，我们来看看当从 999999 到 1000000 时，会发生什么：

```
$ ls mysqlld1-bin.[0-9]*
```

```
mysqlld1-bin.000007 mysqlld1-bin.000011 mysqlld1-bin.000039
mysqlld1-bin.000008 mysqlld1-bin.000035 mysqlld1-bin.1000000
mysqlld1-bin.000009 mysqlld1-bin.000037 mysqlld1-bin.999998
mysqlld1-bin.000010 mysqlld1-bin.000038 mysqlld1-bin.999999
```

你会看到，最后创建的 binlog 文件的次序出现在靠前的两个文件之前。因此使用通配符前有必要检查一下文件名。

由于 binlog 文件通常都很大，你可能并不希望把 binlog 文件的整个内容都输出出来看。

有几个选项可以限制输出结果，从而只输出部分事件。

#### `start-position=bytepos`

转储的第一个事件的字节位置。请注意，如果向 *mysqlbinlog* 提交了多个 binlog 文件，这个位置就是第一个文件的第一个事件的字节位置。

如果事件不是从给定位置开始的，*mysqlbinlog* 仍然会将这个位置解释为事件开始的字节位置，这通常就会产生垃圾输出。 111

#### `stop-position=bytepos`

最后输出的事件的字节位置。如果事件没有在那个位置结束，最后输出的事件就是在 *bytepos* 位置之前的事件。如果给定了多个 binlog 文件，该位置就是最后一个文件最后输出的事件的字节位置。

#### `start-datetime=datetime`

只输出那些时间戳等于或大于 *datetime* 的事件。同样适用于给定多个文件的情况，但不检查事件是否按照它们的时间戳顺序输出。如果某个文件的所有事件都发生在 *datetime* 之前，所有的事件都会被略过。

#### `stop-datetime=datetime`

只输出那些时间戳小于 *datetime* 的事件。这是一个开区间，意味着如果一个事件被标记为 2010-01-24 07:58:32，并且给定 *datetime* 也正好就是这个值，那么不会输出这个事件。

请注意，事件使用语句的开始时间作为时间戳，而二进制日志中的事件是根据提交时间来排序的，有可能出现排在后面的事件的时间戳在它前面的事件的时间戳之前。这个选项使 *mysqlbinlog* 在区间外的第一个事件处停止，所以可能有些事件不会显示，因为它们的时间戳在 *datetime* 之前。

## 读取远程文件

*mysqlbinlog* 工具不仅可以在本地文件系统读取文件，还可以从远程服务器上读取 binlog 文件，其机制与 slave 连接 master 并请求事件一样。这在一些情况下是可行的，因为它不要求机器上的 shell 账户读取 binlog 文件，只需要服务器上有一个拥有 REPLICATION SLAVE 权限的用户即可。

使用 `read-from-remote-server` 选项远程读取 binlog 文件，其参数包括要连接服务器的主机和用户、可选的端口号（如果不是默认值的话）及密码。

从远程服务器读取 binlog 文件时，只需给出 binlog 文件的名称，而不需给出它的全路径。

那么，要远程读取示例 4-18 中的 Query 事件，可使用以下命令（服务器提示输入密码，但是回车显示的不是输出）：

```
$ sudo mysqlbinlog
> --read-from-remote-server
> --host=master.example.com
> --base64-output=never
> --user=repl_user --password
> --start-position=294
> mysqlbin.000038
Enter password:
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=1*/;
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 294
#130608 22:09:19 server id 1  end_log_pos 0  Start: binlog v 4, server v 5.5.31
-0ubuntu0.12.04.1-log created 130608 22:09:19
# at 294
#130608 22:13:08 server id 1  end_log_pos 362  Query thread_id=53 exec_time=0
error_code=0
SET TIMESTAMP=1370722388/*!*/;
SET @@session.pseudo_thread_id=53/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0...
SET @@session.sql_mode=0/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_offset...
/*!\C utf8 *//*!*/;
SET @@session.character_set_client=33,@@session.collation_connection=33...
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
BEGIN
/*!*/;
# at 362
#130608 22:13:08 server id 1  end_log_pos 390  Intvar
SET INSERT_ID=1/*!*/;
# at 390
#130608 22:13:08 server id 1  end_log_pos 472  User_var
SET @`password`:=_utf8 0x2A31353141463642384333413641413039434643434244333...
# at 472
#130608 22:13:08 server id 1  end_log_pos 627  Query thread_id=53 exec_time=0
use `test`/*!*/;
SET TIMESTAMP=1370722388/*!*/;
INSERT INTO employee(name, email, password)
VALUES ('mats', 'mats@example.com', @password)
/*!*/;
```



```
# at 627
#130608 22:13:08 server id 1  end_log_pos 654 Xid = 175
COMMIT/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=0*/;
```

## 读取日志文件的原始二进制文件

*mysqlbinlog* 工具不仅可以用于审查二进制日志，还可以获取 binlog 文件的备份。如果你无法访问机器但却拥有 REPLICATION SLAVE 权限，这时就非常有用。你可以从服务器读取二进制的日志文件，但不会解析这些文件，而仅仅是存储它们。

113

通常，使用 *mysqlbinlog* 获取远程 binlog 文件备份的方法是：

```
mysqlbinlog --raw --read-from-remote-server \
  --host=master.example.com --user=repl_user \
  master-bin.000012 master-bin.000013 ...
```

这里，binlog 文件 *master-bin.000012* 和 *master-bin.000013* 将从当前目录读取并存储。注意，必须同时使用 *--read-from-remote-server* 和 *--raw*。如果只用 *--raw* 而不用 *--read-from-remote-server* 是没意义的，这就相当于使用一个纯文本副本。下面讨论一些有趣的控制选项。关于如何使用 *mysqlbinlog* 做备份有更加详细的描述，参见 MySQL 参考手册。

### *--result-file=prefix*

该选项给出创建写入文件的前缀，这个前缀可以是目录名（带有反斜杠），或者任何其他前缀。默认值是一个空字符串，所以如果不使用这个选项，即将写入的文件与 *master* 上的文件同名。

### *--to-last-log*

一般来说，只有命令行给定的文件才会被传送。但如果提供了这个选项，只需要给定开始传送的二进制日志文件，然后 *mysqlbinlog* 会把后面所有文件都传送出去。

### *--stop-never*

到达最后一个日志文件末尾也不停止，继续等待更多输入。在做即时恢复备份的时候，这个选项很有用。参见第 15 章“备份和 MySQL 复制”一节获取更多关于即时恢复备份的详细内容。

# 解释事件

有时候, *mysqlbinlog* 输出的标准信息并不足以解决问题, 因此必须深入事件的具体细节, 审查其内容。为了处理这种情况, 可以使用 *hexdump* 选项告诉 *mysqlbinlog* 去写事件的实际字节。

在查看事件细节之前, 我们先了解一下二进制日志中数据格式的一些常规规则。

114

## 整数数据

二进制日志的整数字段以小端字节序输出, 因此必须从后往前读取整数字段。例如, 32 位的块 03 01 00 00 代表十六进制数 103。

## 字符串数据

通常字符串数据与长度数据和非空终端字符一起被存储。有时候, 长度数据正好在字符串前面, 有时候它又被存在 post 头中。

本节将涵盖大部分常用事件, 而所有事件的格式细节已经超出了本书的讨论范围。要想了解所有事件及其字段的信息, 请参看 MySQL 内部手册中的“二进制日志”部分。最常见的事件是 Query 事件, 因此我们先来研究它。示例 4-19 展示了这个事件的输出。

示例4-19: 使用 -hexdump 选项时的输出

```
$ sudo mysqlbinlog \
> --force-if-open \
> --hexdump \
> --base64-output=never \
> /var/lib/mysql1/mysqld1-bin.000038
.
.
.
1 # at 496
2 #100123 7:21:33 server id 1 end_log_pos 643
3 # Position Timestamp Type Master ID Size Master Pos Flags
4 # 1f0 6d 95 5a 4b 02 01 00 00 00 93 00 00 00 83 02 00 00 10 00
5 # 203 06 00 00 00 00 00 00 00 00 04 00 00 00 1a 00 00 00 40 |.....|
6 # 213 00 00 01 00 00 00 00 00 00 00 00 00 06 03 73 74 64 |.....std|
7 # 223 04 08 00 08 00 08 00 74 65 73 74 00 49 4e 53 45 |.....test.INSE|
8 # 233 52 54 20 49 4e 54 4f 20 75 73 65 72 28 6e 61 6d |RT.INT0.employee|
9 # 243 65 2c 65 6d 61 69 6c 2c 70 61 73 73 77 6f 72 64 |.name.email.pass|
10 # 253 29 0a 20 20 56 41 4c 55 45 53 20 28 27 6d 61 74 |word....VALUES..|
11 # 263 73 27 2c 27 6d 61 74 73 40 65 78 61 6d 70 6c 65 |.mats...mats.exa|
12 # 273 2e 63 6f 6d 27 2c 40 70 61 73 73 77 6f 72 64 29 |mple.com...passw|
13 # 283 6f 72 64 29 |ord.|
14 # Query thread_id=6 exec_time=0 error_code=0
```

```
SET TIMESTAMP=1264227693/*!*/;  
INSERT INTO employee(name,email,password)  
VALUES ('mats','mats@example.com',@password)
```

前两行和第 14 行是注释，给出了前面讲过的一些基本信息。请注意，使用 `--hexdump` 选项时，通用信息和事件特定的信息被分成两行，而在常规输出结果中则将它们合并。

第 3 行和第 4 行列出了通用头。

#### Timestamp

115

事件的时间戳，整型数，以小端字节序格式存储。

#### Type

事件的类型，单个字节。MySQL 内部手册中给出了一些事件类型，而要获取特定服务器的值需要查看源代码（在 `sql/log_event.h` 文件中）。

#### Master ID

写该事件的服务器的服务器 ID，整型数。比如示例 4-19 中的事件，服务器 ID 是 1。

#### Size

事件的大小（字节数），整型数。

#### Master Pos

同 `end_log_pos` 类似，即该事件之后的事件的开始位置。

#### Flags

与事件相关的通用标记，占 16 个比特位。大多数情况下不使用该字段，但是它存储了 `binlog-in-use` 标记。正如示例 4-19 所示，设置 `binlog-in-use` 意味着二进制日志关闭不正确（在这个例子中，原因是调用 `mysqlbinlog` 之前未刷新日志）。

通用头之后是提交头（post header）和事件体。前面提过，所有事件的完整列表超出了本书的讨论范围，我们将涵盖最重要、最常用的事件：Query 和 Format\_description 日志事件。

### 查询事件的提交头和事件体

Query 事件是到目前为止服务器发出的最常用、最复杂的事件。一部分原因是它必须承载语句执行时所需的大量信息。我们已经讲过，特定事件涉及整型变量、用户变量和随机种子，还有必要提供一些其他信息，这是 Query 事件的一部分。

Query 事件的提交头由 5 个字段组成。回想一下，这些字段都是固定大小的，提交头的

长度由 binlog 文件中的 `Format_description` 事件给定。如果有需要, 以后的 MySQL 版本可以添加额外的字段。

#### Thread ID

表示执行语句的线程 ID, 无符号整型, 4 字节。尽管线程 ID 并不是正确执行语句所必需的, 但是它总是被写入事件。

#### 116 Execution time

从开始执行查询到写入二进制日志所花费的秒数, 无符号整型, 4 字节。

#### Database name length

数据库名的长度, 无符号整型, 1 字节。数据库名存在事件体中, 但其长度在这里给出。

#### Error code

语句执行产生的错误代码, 无符号整型, 2 字节。要有这个字段是因为, 在某些情况下, 即使语句执行失败了, 它们也必须记录到二进制日志中。

#### Status variables length

事件体中保存状态变量的块长度, 无符号整型, 2 字节。这个状态块有时和 Query 事件一起用来存储各种状态变量, 例如 `SQL_MODE`。

事件体包括以下字段, 它们都是可变长度的。

#### Status variables

状态变量序列。每个状态变量由一个单整型加上状态变量值表示。状态变量值的解释和长度都取决于它描述的是哪个状态变量。状态变量不总是存在的, 它们只在必要时添加。例如下面这些状态变量:

##### Q\_SQL\_MODE\_CODE

当语句执行时使用 `SQL_MODE` 的值。

##### Q\_AUTO\_INCREMENT

这个状态变量包含 `auto_increment_increment` 和 `auto_increment_offset` 的值, 假定它们的默认值不为 1。

##### Q\_CHARSET

这个状态变量包含语句执行时连接和服务端所使用的字符集代码和排序规则。

#### Current database

当前数据库的名称, 非空终结符 (null-terminated) 的字符串。请注意, 数据库名的长度在提交头中给定。



#### Statement text

被执行的语句。语句的长度可以从通用头和提交头中计算出来。一般来说，这个语句与原始记录的语句相同，但是在某些情况下，语句在写入二进制日志前被重写。例如，正如你在本章前面所看到的，在指定 `DEFINER` 从句的时候存入触发器和存储过程。

117

### 格式描述事件提交头和事件体

`Format_description` 事件记录了关于 binlog 文件格式、事件格式和服务器的信息。该事件必须在各个版本之间保持鲁棒性（即哪怕 binlog 格式发生变化，它仍然是可解释的），所以对哪些是可变的有一些限制。

最重要的限制之一是 `Format_description` 事件和 `Rotate` 事件的通用头都是固定的 19 字节。也就是说，在通用头中增加新字段来扩展事件是不可能的。

`Format_description` 事件的提交头和事件体包含以下字段。

#### Binlog file version

文件所使用的 binlog 文件格式的版本。对于 MySQL 5.0 版本及其后面的版本来说，这个值是 4。

#### Server version string

服务器版本的信息，字符串型，50 字节。一般来说，这个版本号由三部分数字串组成，加上 build 选项信息，例如 “5.5.31-0ubuntu0.12.04.1”。

#### Creation time

服务器自启动后写入的第一个 binlog 文件的创建时间，即自 1970 年 1 月 1 日 (00:00:00 GMT) 以来的秒数，整型数，4 字节。对于后续服务器写入的 binlog 文件来说，这个字段值为 0。

这样，slave 可以确定服务器已经重新启动，然后 slave 应该重置状态和临时数据。例如，关闭一切活动的事务，并删除所有已经创建的临时表。

#### Common header length

表示 binlog 文件中除了 `Format_description` 和 `Rotate` 事件的所有事件的通用头长度。正如前面所描述的，`Format_description` 和 `Rotate` 事件的通用头的长度是固定的，即 19 字节。

#### Post-header lengths

这是 `Format_description` 日志事件唯一一个长度可变的字段。它是一个整型数组，

其中每个元素是 binlog 文件中的各个事件的提交头大小。该字段的预留长度为 255，因此提交头的最大长度是 254 字节。

## 118 二进制日志的选项和变量

有一组选项和变量用于对二进制日志进行多方面的配置。

有几个选项用于控制诸如 binlog 文件名和索引文件名这样的属性。大多数选项可以同时作为服务器变量处理。有一些选项我们已经在前面提到了，这里将逐个介绍更多细节。

**expire-log-days=days**

binlog 文件需要保留的天数。当二进制日志轮换或服务器重启时，比这个数值旧的那些文件将从文件系统中清除。

这个选项的默认值是 0，即 binlog 文件永远不会被删除。

**log-bin[=basename]**

第 3 章解释过，通过向 *my.cnf* 文件添加 log-bin 选项可以开启二进制日志。这个选项除了可以开启二进制日志，还给出了 binlog 文件的基本文件名，即 “.” 之前的部分。如果还给定了扩展名，则删除它以构成 binlog 文件的基本名。

如果该选项没有给定 *basename*，则基本名默认为 *host-bin*，其中 *host* 是 *pid-file* 选项指定的文件的基本名（即不包含目录和扩展名），通常就是 *gethostname(2)* 中的主机名。例如，如果 *pid-file* 值为 */usr/run/mysql/master.pid*，默认的 binlog 文件名将是 *master-bin.000001*、*master-bin.000002* 等。

*pid-file* 选项的默认值包含主机名，强烈建议为 log-bin 选项赋值。否则如果主机名发生变化，binlog 文件名也会变化（除非 *pid-file* 不是默认值）。

**log-bin-index[=filename]**

给定索引文件名。如果不想把索引文件放在默认位置，该选项很有用。

默认值与 log-bin 中的基本名相同。例如，如果用于创建 binlog 文件的基本名是 *master-bin*，那么索引文件名为 *master-bin.index*。

与 log-bin 选项的情况类似，主机名是索引文件名的一部分，也就是说，如果主机名改变，复制将中断。因此，强烈建议为这个选项赋值。

## 119 log-bin-trust-function-creators

创建存储函数时，可能会创建一些允许在 slave 上进行任意的数据读取和处理的特

殊函数。因此，创建存储函数需要 SUPER 权限。然而，由于存储函数很多时候都非常有用，DBA 相信任何拥有 CREATE ROUTINE 权限的用户不会编写恶意的存储函数。因此，创建存储函数需要 SUPER 权限的要求可能会被废除（当然还是需要 CREATE ROUTINE 权限）。

#### `binlog-cache-size=bytes`

事务缓存在内存中的部分的大小，以字节数计。磁盘存储支持事务缓存，因此如果事务缓存的大小超过这个值，剩余的数据将存入磁盘。

这有可能带来性能问题，因此如果有很多大型事务，增加这个值可以提高性能。

注意，仅仅分配一个非常大的缓冲区并不是一个好主意，因为这意味着服务器的其他部分可用内存变少，这也会导致性能下降。

#### `max-binlog-cache-size=bytes`

该选项用于限制二进制日志中每个事务的大小。大型事务有可能长时间阻塞二进制日志，从而需要其他线程为二进制日志护航，因而造成重大的性能问题。如果事务的大小超过这个值，语句将出错而被中止。

#### `max-binlog-size=bytes`

指定每个 binlog 文件的大小。如果写入语句或事务导致文件大小超过这个值，binlog 文件将被轮换，并写入一个新的空的 binlog 文件。

注意，如果事务或语句超过了 `max-binlog-size`，二进制日志将被轮换，该事务作为一个整体将被写入新的文件。这是因为事务永远不会被分割到不同的 binlog 文件。

#### `sync-binlog=period`

通过 `fdatsync(2)` 函数，指定二进制日志多久写一次磁盘。这个值是每次实际调用 `fdatsync(2)` 时的事务提交数。例如，如果给定值为 1，每次事务提交都要调用 `fdatsync(2)`；如果给定值为 10，则每 10 次事务提交调用一次 `fdatsync(2)`。

值为 0 表示永远不会调用 `fdatsync(2)`，服务器信任操作系统，将二进制日志写磁盘作为常规文件处理的一部分。

120

#### `read-only`

阻止任何客户端进程（除了具有 SUPER 权限的 slave 线程和用户）更改服务器上的任何数据（不包括临时表，它们是可更改的）。这对于 slave 服务器的复制工作非常有用，可以保证连接 slave 的客户端不破坏数据。

## 基于行的复制参数

配置基于行的复制需要以下参数。

### binlog-format

binlog-format 参数可以设置为下面几种模式。

#### STATEMENT

所有语句都使用传统的基于语句的复制。

#### ROW

所有插入或改变数据的语句(即数据操纵语言 DML 语句)都使用基于行的复制。但是,创建表或其他更改模式(schema)的语句(即数据定义语言 DDL 语句)仍使用基于语句的复制。

#### MIXED

这是基于语句的复制的安全版本,MySQL 5.1 及其后的版本推荐使用此模式。在混合复制模式中,服务器以语句的方式将语句写入二进制日志中,如果语句不安全(判断标准请参考前文),则切换为基于行的复制。

该变量还可作为全局服务器变量和会话变量。开始新的会话时,全局变量的值被复制到会话变量中,然后使用这个会话变量确定如何将语句写入二进制日志。

### binlog-max-row-event-size

该参数用于指定何时开始下一个包含行的事件。由于事件在处理时被完全读入内存,该参数粗略控制那些包含行的事件的大小,保证处理行的时候不会消耗过多的内存。

### binlog-rows-query-log-events (MySQL 5.6 中新引入的选项)

该选项使服务器在行事件之前向二进制日志添加一个信息事件(informational event)。这个信息事件包含产生这些行的原始查询。

**121** 注意,由于无法解析的事件会导致 slave 停止,MySQL 5.6.2 之前的 slave 服务器不能解析这个事件,从而停止从 master 的复制工作。也就是说,如果要使用信息事件,需要在启用 binlog-rows-query-log-events 选项之前把所有 slave 都升级到 MySQL 5.6.2 (或之后的版本)。

从 MySQL 5.6.2 开始,也可以为了其他目的添加信息事件,但并不会导致 slave 停机。这些信息事件允许向二进制日志添加信息,这样读取者(包括 slave)就可以使用它们。但这些事件不应该以任何方式更改执行的语义,因此出于安全,无法读取这些信息事件的 slave 会忽略它们。



## 小结

显然，二进制日志涉及很多内容，包括它的使用、构成和技术等。本章我们介绍了这些概念和更多的内容，包括如何控制二进制日志的行为。通过本章的学习，我们对二进制日志的机制有了更深入的了解，以及记录数据变更的重要性。

Joel 打开一封老板发来的无标题的邮件。“我讨厌这样，”他想。Summerson 先生的邮件就像他的任务一样——直截了当。邮件是这么写的：“谢谢你为营销部门恢复数据。期待明天上午的报告。你可以通过电邮发给我。”

Joel 耸耸肩，开始写邮件，认真地写了一个有意义的邮件主题。他不知道应该写多详细，是不是要解释一下二进制日志和 `mysqlbinlog` 工具。他想了一会儿，写下了足够多的细节。“他很可能让我缩减成一个项目符号列表，”Joel 想。这主意好像不错，于是他写了两句总结，并列了一些要点，然后把它们移到邮件顶部。写完后，他把这个邮件发送给了老板。“也许我应该把这些东西保存起来，万一要我讲呢，”他自言自语道。

## 面向高可用性的复制

Joel 正在听 iPod，发现老板就站在他桌前。他摘下耳机说：“对不起，老板。”

Summerson 先生微笑着说：“没关系的，Joel。我需要你想个办法监控复制服务器，保证数据不丢失，宕机时间最短。我们开始听到一些开发人员抱怨当前系统不是很灵活。我能搞定那些开发人员，但是技术支持人员告诉我，如果服务器出错，要花很长时间才能恢复。我希望你优先解决这个问题。”

Joel 点了点头，“没问题，我会着手复制中的负载平衡和恢复问题。”

“很好。看看解决这个问题我们需要做些什么，给我出份报告。”

Joel 目送他的老板离开办公室。“好吧，来看看高可用性这一章怎么说吧，” Joel 一边想，一边打开了他最爱的 MySQL 书。

购买昂贵的可靠的机器，并确保拥有一个良好的 UPS（不间断电源，Uninterrupted Power Supply）防止系统断电，应该就可以得到一个高可用的系统了，对吧？

好吧，高可用性不是那么容易就能实现的。要想搭建一个真正的、持续可用的系统，你必须仔细考虑到任何偶然故障的发生，确保有冗余以处理故障组件。真正的高可用性是指即使在最意想不到的情况下系统都不会停止运行，这样的高可用性系统是非常难以实现的，而且成本很高。

124 实现高可用性的基本原则非常简单，难点在于实现它们。确保高可用性需要做 3 件事：

冗余

如果一个组件出现故障，你必须有一个可替代品。该替代品既可以是闲置的，也可以是当前系统部署中的一部分。

## 应急计划

如果一个组件出现故障，你必须知道接下来该做什么。这取决于哪个组件出现故障，以及为何出现故障。

## 程序

如果一个组件出现故障，你必须能够检测出故障原因，然后迅速有效地执行你的计划。

如果系统中单个组件的故障导致整个系统瘫痪，称为单点故障（single point of failure）。如果系统中存在单点故障，就会严重限制我们实现高可用性的能力。因此，首要目标之一，是要找到这些单点故障，确保我们做了冗余处理。

# 冗余

为了搞清楚哪里需要做冗余，必须明确系统部署中每一个可能出现的故障点。虽然这听起来容易——当然也有一些烦琐和无聊——你需要一定的想象力才能完全找到它们。交换机、路由器、网卡甚至通信电缆都可能发生单点故障。除了架构，电源和物理设备同样重要。还有，部署维护服务呢？如果所有网络管理都是通过单个基于 Web 的接口呢？或者，如果只有一名工作人员知道如何处理某些类型的故障呢？怎么办？

确定这些故障并不意味着你必须完全消除它们。有时，从经济、技术、地理等因素考虑，完全消除也是不可能的，但是知道它们的存在有助于我们做计划。

有些应该考虑或至少需要刻意决定要不要考虑的事情是：组件副本的成本、不同组件发生故障的概率、替代故障组件的时间和修复组件的风险。如果修复组件需要一周的时间，而这段时间内使用备用组件也可能造成单点故障，其实你就是在承担风险，这可能是可接受的（或不可接受的）。

一旦确定了哪里需要冗余，我们需要从两个基本方案中选择：（1）为每个组件保留副本，一旦原先的组件发生故障，副本马上接管；或（2）确保系统有额外的处理能力，一旦组件出现故障时，依然可以处理负载。这并不是二选一的抉择，你可以结合两种方案，备份一些组件，而其他部分使用额外处理能力。

125

表面看起来，最简单的方法是为组件做副本，但是其实耗费很大。你必须时刻准备一个组件副本，还要保持与主要组件同步更新。备份组件的好处是，切换时不会影响性能，而且切换到备用组件通常比系统重建快多了。但是如果在创建备用处理能力时遇到问题，你也只能选择重建系统。

创建备用处理能力让你可以使用全部组件来运行业务，可以处理更高的负载。如果一个组件出现故障，要重建系统才能继续使用剩余组件。重要的是，你需要比常规需求更多的处理能力。

我们通过一个简单的例子来解释原因。有一个 master 处理写任务——实际上应该有两个 master，因为需要冗余——还有一些连接到 master 的 slave，它们的唯一目的就是处理读请求。

如果其中一个 slave 出现故障，系统仍会响应，但系统的处理能力减少。如果你有 10 个 slave，每个 slave 的运行负载是 50%，一个 slave 故障会使每个 slave 的负载增加到 55%，这还很容易对付。但是，如果每个 slave 的运行负载是 95%，那么任何一个 slave 故障会使剩余的 slave 的运行负载增加到 105%，这显然是不可能的。在这种情况下，系统的读能力将会降低，响应时间也会更长。

然而只对一台服务器发生故障做出应对计划还是不够的，还要考虑到多台服务器同时发生故障的可能性及应对方法。继续使用前面的例子，哪怕每台服务器的运行负载是 80%，这个系统还是能够应付一台服务器发生故障的情况。但是，如果两台服务器都发生故障，意味着剩余服务器的负载将上升到 100%，就没有任何能力再应对其他任何突发事件了。如果这种事情一年发生一次，或许还是可控的，但是你必须知道它可能发生的频率。

表 5-1 列出了在配有 100 台服务器的系统中，在不同单点故障概率的情况下，1 台、2 台、3 台服务器分别发生故障的概率。可以看到，在单点故障率是 1% 的情况下，3 台或 3 台以上服务器发生故障的概率是 16%。如果没有做好故障准备，那么一旦真的发生故障，那就真的是麻烦大了。

126



随机变量  $X$  代表发生故障的服务器的数量，通过二项式分布计算其概率为：

$$P(X \geq k) = \binom{n}{k} p^k$$

表5-1：服务器发生故障的概率

单点故障概率	1	2	3
1.00%	100.00%	49.50%	16.17%
0.50%	50.00%	12.38%	2.02%
0.10%	10.00%	0.50%	0.02%

为了避免发生这种情况，你必须密切监控部署系统的负载情况，衡量系统的处理能力，并计算什么时候响应时间开始变慢。

## 计划

只做冗余是不够的，还需要计划当组件发生故障时的应对策略。在之前的例子中，一个



slave 发生故障很容易处理，因为新的连接会被重定向到当前运行的 slave，但是考虑以下情况：

- 怎么处理已存在的连接？只是终止运行然后向用户返回一个错误信息，这并不是一个好主意。通常，在用户和数据库之间有一个应用层，这时应用层必须向另一个服务器重新尝试查询。
- 如果 master 发生故障了怎么办？前面的例子都是假设 slave 发生故障，而 master 也会发生故障。假设有一个冗余的 master（这点将在后面的章节中介绍），我们必须计划好怎样将所有 slave 都移到新 master 上。

本章将会介绍一些技术和拓扑结构，应对 MySQL 服务器发生故障的各种情况。大致考虑三种服务器角色：master 故障、slave 故障和 relay 故障。slave 故障是指用于扩展读操作的 slave 发生故障。同时扮演 master 角色的 slave 称为中继服务器（relay），它们需要特别处理。master 故障是最重要的故障，需要尽快处理，因为 master 不恢复，系统就不可用。

## slave 故障

127

到目前为止，最容易处理的故障是 slave 故障。由于 slave 只用于读查询，告诉负载均衡器 slave 发生故障了，负载均衡器会将新的查询定位到正常工作的 slave。当然必须有足够多的 slave 去应对系统处理能力的降低，除此之外，一个发生故障的 slave 一般不会影响到复制的拓扑结构，也不需要考虑特别的拓扑结构来简化 slave 故障的管理。

当 slave 发生故障时，难免有一些查询已经发送到这个 slave 并在等待查询结果。由于服务器失去连接报错以后，这些查询将会重新递交给正常工作的 slave。

## master 故障

如果 master 发生故障，必须替换它以保证部署系统可以正常运行，而且必须迅速替换。一旦 master 发生故障，所有的写查询都会被中断，所以要做的第一件事就是找一个新的可用的 master，将所有客户端连接到这个新的 master 上。

由于主 master 发生故障，所有 slave 都失去了与 master 的连接，这意味着所有 slave 上都有过时的数据，但是它们仍在运行而且可以响应读查询。

然而，如果有些查询要等待变化到达 slave，那么这些查询会被阻止。有些查询会被写入 slave 上的中继日志，因此最终会在 slave 上执行。不用特殊考虑这些查询。

如果查询还在等待那些 master 崩溃前尚未离开的事件，情况将更加糟糕。在这种情况下，必须确保这些查询被处理。通常是报告查询失败，因此用户必须重新发出查询请求。

## relay 故障

对中继服务器 (relay) 的故障, 需要特殊处理。如果它们发生了故障, 剩余的 slave 必须重定向到其他中继服务器或 master。由于添加中继服务器就是为了减轻 master 的负载, 有可能出现以下情况: master 无法处理某个中继服务器上的所有 slave 的负载。

## 灾难恢复

在高可用性的世界里, “灾难”并不是指地震或者洪水, 而是指电脑出了严重的问题, 不单单指发生故障的机器。

128 ▸ 典型的例子是数据中心断电——不见得是整个城市断电, 可能只是数据中心所在的大楼断电。

灾难的本质在于, 很多故障同时发生, 在单个数据中心冗余多个服务器副本已经无法处理这种情况。因此, 有必要将数据安全保存在另一个物理位置。此外, 为不同办事处创建不同组件, 是很多公司获取高可用性的常用措施, 即便是相对较小的公司也会这么做。

## 方法

消除了所有单点故障, 确保了系统有足够的冗余, 并且为每个紧急事件都做好了计划, 现在应该准备好进行最后一步了。

除非支配正确, 否则所有的资源和精心准备都是无用的。通常, 对一个只有几个服务器的小型网站, 你可以不做规划。但是随着服务器数量的增加, 自动部署将变得很有必要——如果你的生意很成功, 服务器的数量会增加很快。

如果从一开始就计划做自动化, 情况就好多了——以后随着业务增长, 你将会忙于处理其他事情, 可能没有时间去建立必要的自动化支持。

我们已经讨论过一些基本过程, 但是以下任务还需要一些准备工作。

### 添加新的 slave

在扩大规模时创建新的 slave 是运行大规模网站的基础。创建新 slave 的方法有很多。它们基本上都是这个套路: 对现有的服务器 (通常是 slave) 进行快照, 在一个新的服务器上恢复快照, 然后从适当的位置启动复制。

当然, 对服务器做快照的时间, 会影响启用新 slave 的速度。如果备份时间太长, master 上可能已经发生了很多新的变化, 这意味着新的 slave 要花更多的时间同步。因此, 快照时间很重要。图 5-1 展示了 slave 同步的快照时间。可以看出, 当 slave

停下来做快照时，更新开始累积，使得显著更新（纵轴）增加。一旦 slave 重启，它开始应用这些显著更新，从而导致显著更新减少。

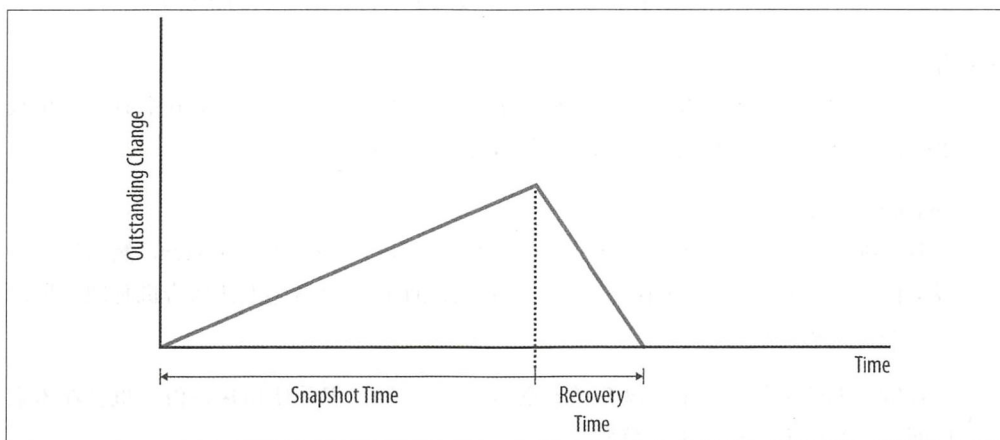


图5-1: 获取快照时的显著更新

获取快照有以下几种方法。

129

#### 使用 *mysqldump*

使用 *mysqldump* 是安全的，但是速度很慢。如果使用 InnoDB 表，通过 *mysqldump* 选项可以得到一致性快照，意味着你不必先把服务器关掉。还有一些选项可以获取 master 和 slave 的快照位置，这样复制就能够从正确的位置开始。

#### 复制数据库文件

这个方法比较快，但是要求在复制文件之前先将服务器离线。此外，还需要管理一些位置，即 *mysqldump* 获取的那些正确启动复制的位置。

#### 使用在线备份方法

有几种方法，例如 MySQL Enterprise Backup 和 XtraBackup。

#### 使用 LVM 获取快照

在 Linux 系统中，可以通过逻辑卷管理器 (LVM) 得到卷快照。不过需要事先做些准备，创建一个特殊的 LVM 卷。与复制数据库文件一样，这个方法也需要自己管理复制位置。

#### 使用文件系统快照的方法

例如，Solaris ZFS 内置支持快照功能。这是快速创建备份的技术，但是和上面提到的其他技术相似（*mysqldump* 除外），需要自己管理复制位置。

如果恢复的时候要使用一个不同的引擎，就必须用 *mysqldump*。其他方法都要求备份和恢复使用一样的引擎。

创建新 slave 的技术已经在第 3 章讲过了，第 15 章将介绍不同的备份方法。

#### 从拓扑结构中删除 slave

只需要通知负载均衡器 slave 已经不存在了，就可以删除 slave。第 6 章有一个负载均衡器的例子，其中包含了添加和删除服务器的方法。

#### 切换 master

一般的维护方式是：将连接在 master 上的所有 slave 切换到副 master，然后通知负载均衡器原来的 master 不存在。这个过程可以而且应该不产生任何宕机时间，所以应该不会影响常规操作。

一种解决办法是使用 slave 提升（本章后面会描述），但是使用热备份（也会在本章后面描述）的方法可能更简单些。

#### slave 故障处理

slave 早晚会出现故障的——这只是频率问题。处理 slave 故障是任何部署工作的例行公事。要做的只是：检测到 slave 不存在了，然后把它从负载均衡器的池中删除，这将在第 6 章讲述。

#### master 故障处理

如果一个 master 突然出现故障，必须检测故障，并把所有 slave 都转移到一个备用 master 上，或者选择一个 slave 将其提升为新的 master。稍后本章将描述这些技术。

#### 升级 slave

通常将 slave 升级到新版本不会出现问题。不过把 slave 从系统中孤立出来进行升级，需要将它从负载均衡器中移除，并需要通知其他系统这个 slave 不存在了。

#### 升级 master

升级 master 首先要升级所有的 slave，这样它们才能从 master 读取全部复制事件。通常在升级 master 时，可以使用备用 master，或者选择一个 slave 将其提升为 master。

## 热备份

做服务器副本最简单的拓扑结构就是热备份（hot standby），如图 5-2 所示。热备份是一个专用服务器，它是主 master 的副本。热备份服务器以 slave 的方式连接到 master，以读取和应用所有更新。这种配置通常称为主 - 备份配置（primary-backup configuration），



其中“主”是指 master，“备份”是指备用服务器。可以有多个热备份。

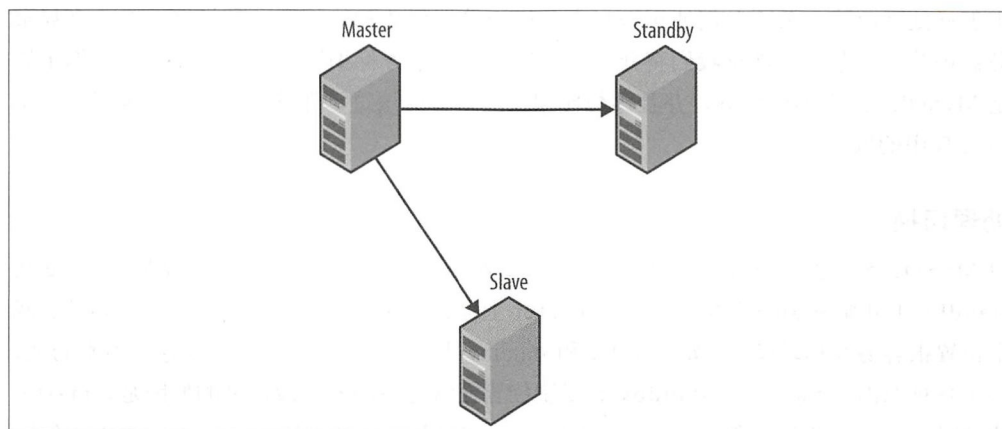


图5-2: 具有一个热备份的master

故障是不可避免的，至少在你运行一个大型部署系统时是这样的。服务器会不会挂掉不是问题，问题是什么时候挂掉，挂掉的频率有多高。这种拓扑结构的思想是，由于热备份提供一个完全一致的 master 副本，如果主 master 挂了，所有的客户端和 slave 可以切换到热备份上继续运行。运行基本不会出错，热备份为我们修复和替换主 master 提供了机会。修复 master 以后，需要让它重新工作，要么将它设置为热备份，要么把所有的 slave 再重定向回来。方法有很多，但现实并不总是那么美好。

首先，我们考虑第一种情况：当主 master 还在运行的时候，切换到热备份，如图 5-3 所示。

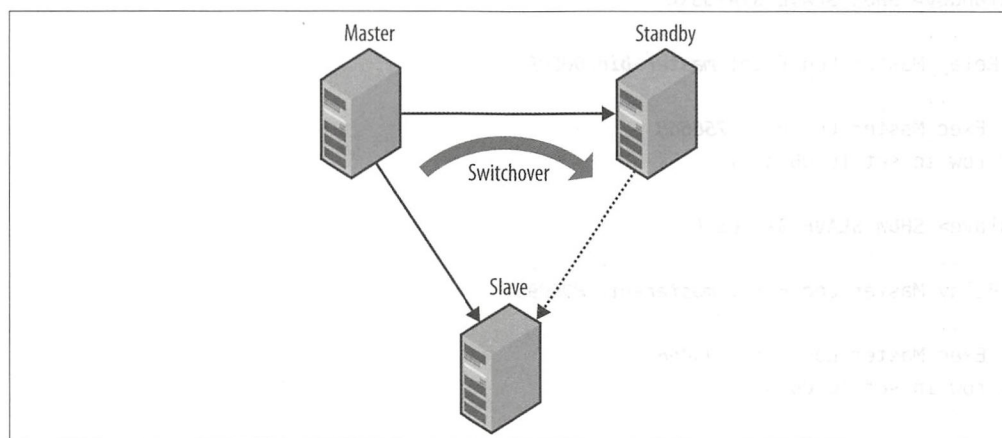


图5-3: 从正在运行的master切换到备份服务器

MySQL 5.6 引入了全局事务标识符 (global transaction identifiers) 的概念, 极大地简化了故障处理问题。但是, 由于 MySQL 5.6 比较新, 所以本节讲述 MySQL 5.6 之前的版本怎样处理故障。关于如何使用全局事务标识符处理故障的问题, 参见第 8 章 “全局事务标识符” 一节, 在那里我们将介绍如何使用全局事务标识符配置服务器。如果你用的是 MySQL 5.6 之前的版本, 并且你想使用全局事务标识符, 那就需要自己实现了, 参见附录 B 中的例子。

## 132 处理切换

对 MySQL 5.6 之前的版本, 切换到备份服务器 (standby) 的主要挑战是: slave 从 standby 上开始复制的位置, 同它在 master 上停止复制的位置, 要完全一致。如果这两个位置很容易互相转换, 比如, master 和 standby 上的位置一样, 就不是问题。不幸的是, 出于各种原因, master 和 standby 的复制位置通常是不同的。最常见的原因是, master 启动时 standby 没有连到 master 上。即使 standby 从一开始就连上 master, 也无法保证事件写入 standby 的二进制日志的方式, 与它写入 master 的二进制日志的方式相同。

执行切换的基本思路是: 在完全相同的位置停止运行 slave 和 standby, 然后把 slave 重定向到 standby。由于 standby 在停止位置之后没有任何变更, 只需确定 standby 的 binlog 位置, 然后让 slave 从那个位置启动。这个任务必须手动执行, 因为简单地停止 slave 和 standby 无法保证它们之间是同步的。

为此, 同时停止 slave 和 standby, 并比较它们的 binlog 位置。由于这两个位置指的是同一个 master 上的两个位置 (slave 和 standby 连接的是同一个 master), 所以检查位置只需要按照字典顺序比较文件名和字节位置即可:

```
standby> SHOW SLAVE STATUS\G
...
Relay_Master_Log_File: master-bin.000096
...
Exec_Master_Log_Pos: 756648
1 row in set (0.00 sec)
```

```
133 slave> SHOW SLAVE STATUS\G
...
Relay_Master_Log_File: master-bin.000096
...
Exec_Master_Log_Pos: 743456
1 row in set (0.00 sec)
```

在这个例子中, standby 在 slave 前面, 它们位于同一个文件, 但是 standby 的位置是 756648, 而 slave 的位置是 743456。所以, 记下 standby 的位置 (即 756648), 然后启

动 slave 直到它赶上 standby。为了确保 slave 赶上 standby，然后在正确的位置停止，使用 START SLAVE UNTIL 命令，就像我们之前停止报表 slave 那样：

```
slave> START SLAVE UNTIL
-> MASTER_LOG_FILE = 'master-bin.000096',
-> MASTER_LOG_POS = 756648;
Query OK, 0 rows affected (0.18 sec)

slave> SELECT MASTER_POS_WAIT('master-bin.000096', 756648);
Query OK, 0 rows affected (1.12 sec)
```

现在 slave 和 standby 的位置完全一样了，一切就绪，使用 CHANGE MASTER TO 命令切换到 standby，把 slave 连接到 standby 并启动它。但是，应该指定什么位置呢？因为 master 上记录停止运行点的文件和位置与 standby 在同一点上的文件和位置是不同的，所以在 master 记录变更的时候需要同时获取 standby 的记录。为此，在 standby 上运行 SHOW MASTER STATUS 命令：

```
standby> SHOW MASTER STATUS\G
***** 1. row *****
      File: standby-bin.000019
      Position: 56447
      Binlog_Do_DB:
      Binlog_Ignore_DB:
1 row in set (0.00 sec)
```

现在你可以用正确的位置，把 slave 重定向到 standby：

```
slave> CHANGE MASTER TO
-> MASTER_HOST = 'standby.example.com',
-> MASTER_PORT = 3306,
-> MASTER_USER = 'repl_user',
-> MASTER_PASSWORD = 'xyzyz',
-> MASTER_LOG_FILE = '
standby-bin.000019',
-> MASTER_LOG_POS = 56447;
Query OK, 0 rows affected (0.18 sec)
slave> START SLAVE;
Query OK, 0 rows affected (0.25 sec)
```

◀ 134

如果反过来，slave 在 standby 前面，只需把前面步骤中的 slave 和 standby 角色对换即可。这种情况可能发生，因为 master 在运行过程中，有些变更可能没有发给 slave 或 standby。在下一节中，我们将讨论怎样处理 master 意外停止的情况，因而无法将变更发给 slave 或 standby。

## 使用 Python 处理切换

示例 5-1 给出了从一个 slave 切换到另一个 master 的 Python 代码。replicate\_to\_position 函数告诉服务器从 master 的指定位置读。当程序返回时，slave 正好在这个位置停止。switch\_to\_master 函数将 slave 定向到新的 master。该程序假定 slave 和新 master 都连接到同一个原始 master 上。否则，位置就没有可比性，程序抛出异常。该程序可以显式给定 master 上的位置，而无须计算。稍后实现故障转移的时候需要用到这个位置。

示例5-1：切换到新master的程序

```
from mysql.replicant.commands import (
    fetch_slave_position,
    fetch_master_position,
    change_master,
)

def replicate_to_position(server, pos):
    server.sql("START SLAVE UNTIL MASTER_LOG_FILE=%s, MASTER_LOG_POS=%s",
               (pos.file, pos.pos))
    server.sql("SELECT MASTER_POS_WAIT(%s,%s)", (pos.file, pos.pos))

def switch_to_master(server, standby, master_pos=None):
    server.sql("STOP SLAVE")
    server.sql("STOP SLAVE")
    if master_pos is None:
        server_pos = fetch_slave_position(server)
        standby_pos = fetch_slave_position(standby)
        if server_pos < standby_pos:
            replicate_to_position(server, standby_pos)
        elif server_pos > standby_pos:
            replicate_to_position(standby, server_pos)
        master_pos = fetch_master_position(standby)
    change_master(server, standby, master_pos)
    standby.sql("START SLAVE")
    server.sql("START SLAVE")
```

## 135 双主结构

经常提到的一个高可用性配置是双主（dual masters）结构，即两个 master 互相复制，保持同步。双主结构是对称的，用起来非常简单。将故障切换到备份 master 上不需要重新配置主 master，反过来，备用 master 故障时再切换回主 master 也很简单。

服务器可以是主动的（active）或者被动的（passive）。如果服务器是主动的，是指服务器接受写操作，这些写操作可以通过复制传播到其他地方。如果服务器是被动的，是指



它不接受写，只是跟随主动的 master，一旦主动 master 发生故障可以代替它。

使用双主结构，根据目的不同，有两种不同的配置：

#### 主动 - 主动 (active-active)

在这个配置中，写操作同时到达两个服务器，然后将变更发送给对方。

#### 主动 - 被动 (active-passive)

在这个配置中，负责处理写操作的称为主动 master，另外一个称为被动 master，与主动 master 保持同步。

这就跟热备份配置差不多，但由于它是对称的，很容易在两个 master 之间来回切换，轮流成为主动 master。

注意，这个配置不需要被动 master 响应查询。稍后你将看到，在有些方案中，被动 master 其实是一个冷备份。

并不是说一定要用复制来同步服务器，其他技术也可以达到这个目的。有些技术可以支持主动 - 主动 masters，而有些技术只能支持主动 - 被动 masters。

主动 - 主动双主结构的最常见用途是，使不同的用户集（比如分散在世界各地的分区办事处）能够访问地理位置较近的服务器。这样，用户可以使用本地服务器，变更会被复制到另一个 master 以便两个 master 保持同步。由于事务在本地被提交，系统响应更快。理解事务在本地提交很重要，这意味着两个 master 不是一致的（例如它们拥有的信息不一样）。一个 master 上提交的变更最终会传播到另一个 master，但是在此之前，两个 master 上的数据是不一致的。

有两个主要后果需要注意：

- 如果两个 master 都更新了同样的信息，比如，不小心同时向两个 master 添加用户，这两个变更之间将会产生冲突，可能会导致复制停止。
- 如果在两个 master 不一致的时候发生了系统崩溃，有些事务将会丢失。

只允许写一个 master 可以从一定程度上避免变更冲突问题，从而使另一个 master 成为被动的 master。这就是主动 - 被动配置，其中主动的服务器又称主服务器 (primary)，被动的服务器又称备用服务器 (secondary)。

使用异步复制不可避免的结果是服务器崩溃时会丢失事务，当然这取决于应用程序，不一定是一个严重的问题。使用 MySQL 5.5 的新功能——半同步复制 (semisynchronous replication)，可以限制事务丢失的数量。半同步复制的原理是：提交事务的线程会被阻

塞，直到至少有一个 slave 确认收到这个事务。由于事务提交到存储引擎后事件才会发给 slave，所以事务的丢失数量可以控制到最多每线程 1 个。

与主动 - 主动方法相似，主动 - 被动配置也是对称的，因此很容易在主 master 和备份服务器之间来回切换。根据不同的处理镜像的方法，也可能使用被动 master 完成一些诸如升级服务器这样的管理性任务，然后在升级完成后使用升级服务器作为主动 master，而不会产生任何宕机时间。

使用主动 - 被动配置的一个重要问题是，解决两台服务器同时成为主 master 的风险问题，又称为裂脑综合征 (split-brain syndrome)。如果网络连接短时间丢失，而在这段时间内 secondary 主动将自己提升为 primary，但是随后，原来的 primary 重新联机，这时就会产生这个问题。如果变更发生在两台服务器同时作为 primary 的时候，就有可能发生冲突。如果用的是共享磁盘，两个服务器同时写磁盘就可能数据库发生一些“有趣”的问题（比如灾难性的、难以定位的问题）。也就是说，两个运行中的 MySQL 服务器不能共享同一个数据目录，所以需要确保任何时候至多只有一个 MySQL 服务器使用当前数据目录（更多内详细容参考本章后面的“共享磁盘”一小节）。

为了阻止这种情况发生，最简单最常用的方法是确保那个挂掉的服务器真的不在运行，通过一种称为 STONITH（名字有点怪，叫“打爆另一个节点的头”）的技术实现。实现方式有很多种，例如连接到服务器然后执行 kill -9（如果服务器可达的话），关闭网卡隔离服务器，或者关掉机器的电源等。如果服务器真的不可达，下次服务器又能访问的时候要使用“毒药丸”，让它“自杀”。

## 共享磁盘

一个简单的双主结构如图 5-4 所示，其中两个 master 通过一个共享磁盘结构（如 SAN，存储区域网络）连接。在这种方法中，两台服务器连接到同一个 SAN，并使用相同的文件。其中一个 master 是被动的，它不会写任何文件，而主动的 master 则正常运行。如果主服务器发生故障，备用服务器随时接替。

这种方法的优点是，由于 binlog 文件存储在共享磁盘上，所以不需要转换 binlog 的位置。两台服务器就是彼此真正的镜像，只不过在两个不同的机器上运行而已。这意味着从主 master 到备份服务器的切换速度很快。slave 不需要将位置翻译给新 master，只需要记住 slave 停止的位置，执行 CHANGE MASTER 命令，然后再次启动复制。

用这个方法进行故障转移时，必须执行表恢复，因为故障发生时变更可能只做了一半。在这种情况下，每个存储引擎的行为都不一样。例如，InnoDB 必须从事务日志执行常规恢复，与崩溃时的处理一样；而 MyISAM 需要先修复表才能继续操作。这两个选择里

面 InnoDB 更好一点，因为比起修复 MyISAM 表，恢复的速度快很多。

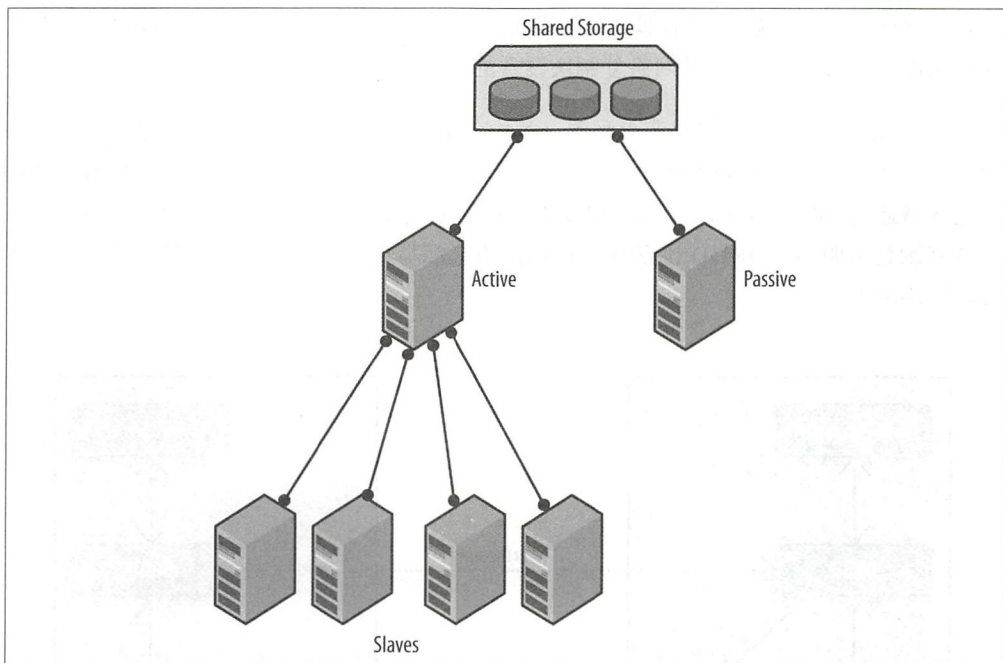


图5-4：使用共享磁盘的双主结构

还要考虑预热缓存的时间，这也可能比较费时。

注意，位置使用的是主服务器的服务器 ID，但却代表备用服务器上的位置，因为这两个服务器使用相同的文件，而且备用服务器是主服务器的镜像。由于位置包含服务器 ID，用户的任何错误也会在这里反映出来，例如传递了一个不是主 master 镜像的 master。

使用共享磁盘建立双主结构取决于使用何种共享存储解决方案，这超出了本书讨论的范围。

使用共享存储的问题是：由于两个 master 使用相同的文件存储数据，在被动 master 上执行管理性任务时必须非常小心。重写配置文件，哪怕是失误，都可能是灾难性的。强制其中一个服务器是只读的仍然不够，因为即使服务器处于只读模式还是会写文件（例如 InnoDB 会写文件）。

◀ 138

处理裂脑综合征问题依赖于使用的共享磁盘解决方案，这超出了本书讨论的范围。例如，SCSI 支持服务器预留磁盘，服务器发现磁盘被另一个服务器预留，从而意识到自己真的不再是 primary，就把自己离线。



## 使用 DRBD 复制磁盘

“Linux 高可用性项目” (<http://www.linux-ha.org/>) 里面有很多维护高可用性系统的有用工具。这些工具大都超出了本书讨论的范围, 但有一个工具很有意思: DRBD (分布式复制块设备), 这是一个通过网络复制块设备的软件。

图 5-5 所示的是一个典型的双节点配置, 其中 DRBD 用于复制磁盘到备用服务器。在每个节点上各自创建一个 DRBD 块设备, 负责把数据写到物理磁盘。这两个 DRBD 进程通过网络通信, 确保 primary 上的任何变更都被复制到 secondary。对 MySQL 服务器来说, 设备复制是透明的。DRBD 设备从外观和行为上都与正常磁盘一样, 所以服务器不需要做特别的配置。

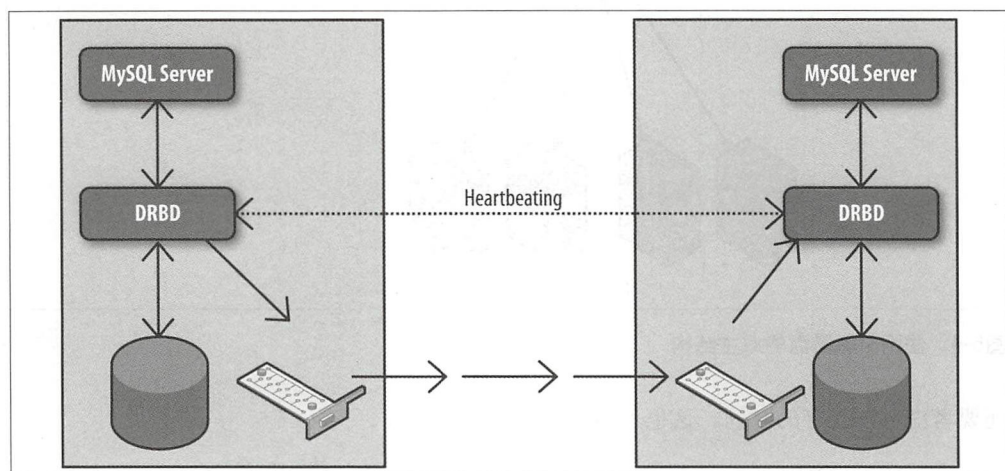


图5-5: 使用DRBD复制磁盘

只能在主动 - 被动配置中使用 DRBD 技术, 也就是说, 被动磁盘完全不能被访问。与前面介绍的共享磁盘解决方案和稍后描述的双向复制实施方案不同, 被动的 master 根本无法访问。

与共享磁盘解决方案类似, DRBD 不需要转换两个 master 之间的位置, 因为它们共享相同的文件。但是, DRBD 技术切换到备用服务器的速度比共享磁盘方案慢。

无论共享磁盘还是 DRBD, 都需要在服务器联机之前恢复数据库文件。由于 MyISAM 表的恢复成本相当高, 建议数据库表使用恢复性能较高的事务性引擎。实践证明 InnoDB 是一个有效的方案, 但是花时间研究其他可选的事务性引擎, 也是值得的。

由于 mysql 数据库对 MyISAM 表的使用限制比较严格, 一般来说, 应该避免在正常运行



过程中对这些 MyISAM 表做不必要的变更。当然，执行管理性任务时这是不可避免的。

相比共享磁盘方案，DRBD 的优势之一是：对于共享磁盘方案来说，磁盘可能带来单点故障。如果共享磁盘阵列的网络连接断了，服务器根本就不会工作。与之不同的是，复制磁盘意味着数据在两个服务器上都可有，从而降低了完全失效的风险。

此外，DRBD 还内置了裂脑综合征问题的解决方案，可以配置为自动恢复。

## 双向复制

如果在主动 - 被动双主结构中使用双向复制，与前面介绍的热备份方案没有显著区别。但是，与其他双主方案不同，双向复制可以有主动 - 主动配置，如图 5-6 所示。

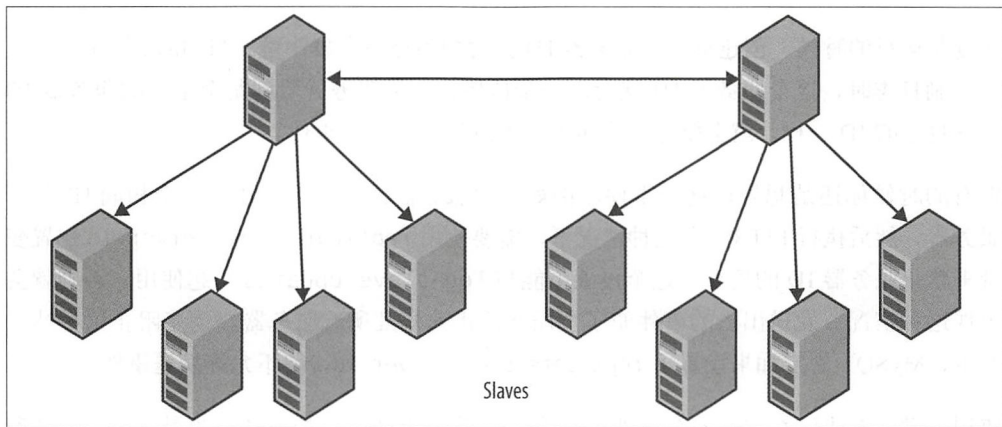


图5-6: 双向复制

虽然有些人对这个方案有争议，但主动 - 主动配置确实有其用途。一个典型的例子是，当两个（或更多）办事处访问同一个数据库上的本地信息（例如，销售数据或员工数据），每个办事处都希望数据库的响应速度快，同时数据在这两个地方都是可用的。在这种情况下，对于每个办事处而言，这些数据其实是本地的：每个销售人员通常都只操作自己的销售业绩数据，一般很少更改其他销售员的数据。

配置双向复制的步骤如下：

1. 确保两台服务器有不同的服务器 ID。
2. 确保两台服务器具有相同的数据（并且在复制启动之前两个系统都没有变更）。
3. 创建一个复制用户，在两台服务器上准备复制（使用第 1 章中的知识）。
4. 在两台服务器上启动复制。



进行双向复制时，要事先确定复制中没有冲突。如果两台服务器更新同一部分数据，不管你有没有发现，都会产生冲突。如果幸运的话，复制会在发生冲突的语句处停止，但不能指望这个。如果想要创建一个高可用性系统，应该确保在应用层两台服务器不会同时更新相同的数据。

即使数据是分区的（例如前面的例子，两个办事处的数据分布在不同的地点），也要确保数据不会在错误的服务器上发生偶然变更，这点是至关重要的。

在这种情况下，应用程序仅在本地进行数据更新是不够的，还要连接负责员工信息的服务器，并在那里进行数据更新。

如果想把 slave 连接到其中一个服务器，要启用 `log-slave-updates` 选项。另一个 master 其实也是作为 slave 连接到这个服务器的，那么问题来了：如果服务器发出去的事件又返回这个服务器，会发生什么？

在复制运行的时候，创建事件的服务器 ID 会附加到每一个事件里。当 slave 把事件写入二进制日志时，这个服务器 ID 就会进一步传播。如果服务器看到某个事件的服务器 ID 与它自己的 ID 一样，就会跳过这个事件，继续处理下一个事件。

但有的时候你还是想处理这个事件。例如，旧服务器没有了，创建了一个相同 ID 的新服务器，然后执行 PITR。在这种情况下，需要使用 `replicate-same-server-id` 配置变量来禁止服务器 ID 的检查。这个设置不能与 `log-slave-updates` 一起使用，否则就会出现这种情况：发送出去的事件兜了个圈然后迅速搞乱所有服务器。为了阻止这种情况发生，MySQL 规定如果设置了 `replicate-same-server-id`，就不允许发送事件。

使用主动 - 主动配置需要安全地处理冲突。目前为止最简单的方法，也是主动 - 主动配置唯一推荐的方法，是保证不同的主动服务器写不同的区域。

142 方案之一是为不同的 master 分配不同的数据库（或者不同的表）。示例 5-2 展示了使用两张表的配置，每个表由不同的 master 更新。为了更好地查看分割数据，我们创建了一个视图合并两个表。

示例 5-2：对不同的办事处使用不同的表

```
CREATE TABLE Employee_Sweden (  
    uid INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(20)  
);
```

```
CREATE TABLE Employee_USA (  
    uid INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(20)  
);
```

```
-- 这个视图用于同时读取两张表
CREATE VIEW Employee AS
  SELECT 'SWE', uid, name FROM Employee_Sweden
  UNION
  SELECT 'USA', uid, name FROM Employee_USA;
```

对于具备天然分割性的数据，这个方法最适用了。例如，不同的办事处有不同的表存储本地数据，只有做报表的时候才需要把数据合并。这看起来挺简单，但是以下问题使表的使用和管理变得复杂。

#### 对不同的表进行读写

根据视图的定义，它是不可更新的。写操作必须直接进入真正的表中，而读操作则可以选择从视图或者直接从表读取。

因此，需要引入应用程序逻辑将读和写操作分割到不同的表。

#### 准确数据和当前数据

由于两张表被不同的站点管理，因此同时更新两张表会导致系统暂时进入这样的状态：两个服务器都拥有对方服务器上不可用的信息。如果此时做快照，这个数据就不准确。

如果要求信息准确，就需要适当的方法。这些方法是高度依赖于应用程序的，这里不进行过多介绍。

#### 优化视图

利用视图创建结果集有两种方法。第一个方法叫 MERGE，即视图被适当展开，就像 SELECT 查询一样可以被优化和执行。第二个方法叫 TEMPTABLE，即构造一个临时表，然后填充数据。

◀ 143

如果服务器使用 TEMPTABLE 视图，它会运行得很糟糕，而 MERGE 视图类似于对应的 SELECT。MySQL 使用 TEMPTABLE，任何时候，视图的定义都没有指出视图的行和基础表中的行有简单的一一映射（例如，视图定义包括 UNION、GROUP BY、子查询或聚合功能），因此，对视图做仔细的设计是获得良好性能的要素。

在这两种情况下，必须考虑使用视图进行报告的影响，因为这可能会影响性能。

如果给每一个服务器分配单独的表，不会发生任何冲突，因为更新完全是隔离的。但是，如果所有的站点都要更新同一个表，就必须换个模式（scheme）。

MySQL 服务器设了两个特殊的服务器变量用于处理这种情况，即：

`auto_increment_offset`

这个变量设置任何 `AUTO_INCREMENT` 列的起始值，即第一个插入表中的行的 `AUTO_INCREMENT` 字段的值。后续行的值则通过 `auto_increment_increment` 计算而得。

`auto_increment_increment`

这是一个增量值，用于计算下一个 `AUTO_INCREMENT` 列的值。



这两个变量都有会话和全局版本，对服务器上的所有表生效，而不仅是刚刚创建的表。每当向含有 `AUTO_INCREMENT` 列的表插入新行时，通过以下方式计算本序列的下一个值：

$$\text{value}_N = \text{auto\_increment\_offset} + N * \text{auto\_increment\_increment}$$

注意：下一个值并不是由表中最后一个值加 `auto_increment_increment` 计算而得的。

在前面的例子中，`auto_increment_increment` 可以确保表中新添加的行的值能够从不同的数字序列获得，这取决于服务器使用什么序列。比如，第一个服务器使用序列 1, 3, 5...（即奇数），而第二个服务器使用序列 2, 4, 6...（即偶数）。

144 ▢ 继续以示例 5-2 为例，示例 5-3 中使用了上述两个变量，保证在向 `Employee` 表插入新员工时，两个服务器用的是不同的 ID。

示例 5-3：两个服务器写同一个表

-- 在任意一个服务器上创建公共表

```
CREATE TABLE Employee (  
    uid INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(20),  
    office VARCHAR(20)  
);
```

-- 设置第一个 master

```
SET GLOBAL AUTO_INCREMENT_INCREMENT = 2;  
SET GLOBAL AUTO_INCREMENT_OFFSET = 1;
```

-- 设置第二个 master

```
SET GLOBAL AUTO_INCREMENT_INCREMENT = 2;  
SET GLOBAL AUTO_INCREMENT_OFFSET = 2;
```

这种模式能够处理向表中插入新条目，但是如果数据被更新，一定要保证更新语句发送给了正确的服务器（即负责员工表的服务器）。否则，数据可能是不一致的。如果更新





没有正确执行，通常不会导致 slave 停止，而是继续复制，从而导致两个服务器上的数据不一致。

例如，如果第一个 master 执行了下面的语句：

```
master-1> UPDATE Employee SET office = 'Vancouver' WHERE uid = 3;  
Query OK, 1 rows affected (0.00 sec)
```

与此同时，同一行数据也在第二个服务器被更新，语句如下：

```
master-2> UPDATE Employee SET office = 'Paris' WHERE uid = 3;  
Query OK, 1 rows affected (0.00 sec)
```

结果是第一个 master 将员工信息放在巴黎，而第二个 master 将员工信息放在温哥华（注意：这个顺序是可调换的，因为每一个服务器会更新完自己之后再更新另一个服务器的语句）。

发现并且阻止这种不一致是很重要的，因为随着时间的推移，它们会传播并产生更多的不一致。由于基于语句的复制根据两个服务器上的数据执行语句，因此一个不一致会导致更多的不一致发生。

如果按照前面的方法仔细隔离两个服务器的变更，然后复制行的变更，这样两个 master 就一致了。

如果用户需要使用不同服务器上的表，防止不一致最简便的方法是设置用户权限，使用户不会不小心更改其他服务器上的表。但是，这个方法并非总是可行的，无法防止刚刚所描述的情况。

## 提升 slave

145

到目前为止，我们描述的方法适用的前提是：master 正常运行，并且能够在切换前同步备份服务器和 slave。但是，如果 master 突然死机了呢？由于所有 slave（包括备用服务器）的复制已经停止，你无法知道每个 slave 上有什么。如果备用服务器在所有 slave 的前面，那就没问题：在每个 slave 上重新运行复制，直到备用服务器的位置处停止。master 上已经发生却尚未发送给备用服务器的变更将会丢失。我们会单独讨论这种情况下如何恢复 master。

如果备用服务器滞后于任何一个 slave，就不能使用备用服务器作为新的 master，因为 slave 知道的比备用服务器更多。实际上，如果从 master 复制事件最多的那个 slave（即“知道的比较多”）就是 master，那就更好了！



这便是提升 slave 处理 master 故障的方法：不是保持一个专门的备用服务器（当然也就没有最佳候选备用），而是确保任何一个连接到 master 的 slave 都能够被提升为 master，并且从 master 故障的位置接管。选择“知道最多”的 slave 作为新的 master，确保任何其他 slave 都没有新 master 知道得多，将它们连接到新的 master，然后从新 master 读取事件。

但是，还有一个重要问题需要解决：同步所有 slave 与新的 master，确保没有任何事件丢失或重复。这时候的问题是：所有 slave 都需要从新 master 读取事件。

## 提升 slave 的传统方法

在研究最终解决方案之前，让我们先来看看提升 slave 的传统推荐做法。这个方法可以很好地说明问题。

图 5-7 展示了典型的一个 master 和多个 slave 的配置。

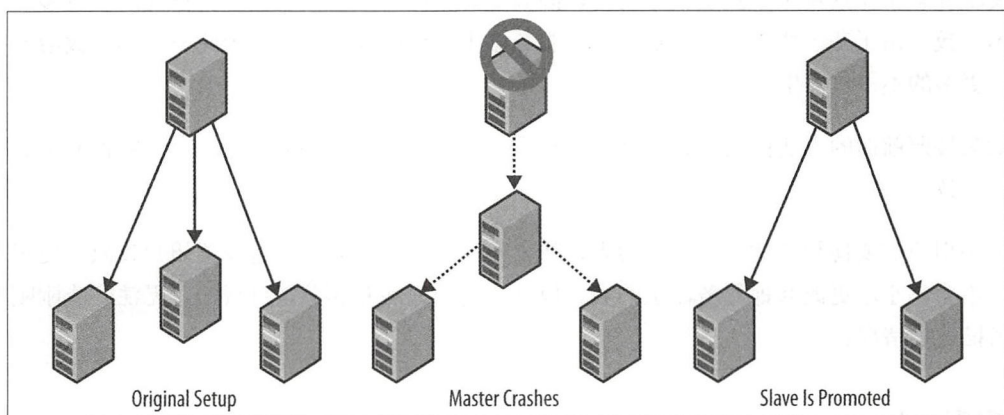


图5-7：提升slave替代故障的master

传统的 slave 提升方法有以下要求：

- 每个可提升的 slave 必须有一个复制用户账户。
- 每个可提升的 slave 运行时必须使用 `log-bin` 选项，即启动二进制日志。
- 每个可提升的 slave 运行时必须不使用 `log-slave-updates` 选项（稍后你自然会明白为什么）。

146 ▢ 假定最初配置如图 5-7 所示，然后 master 发生了故障。将 slave 提升为新 master 的步骤如下：



1. 使用 `STOP SLAVE` 停止 slave。
2. 使用 `RESET MASTER` 重置即将成为新 master 的 slave。slave 将以新 master 的身份启动，其他连接的 slave 从提升的那个时刻开始读取事件。
3. 使用 `CHANGE MASTER` 将其他 slave 连接到新的 master 上。由于重置了新 master，可以从二进制日志的起点开始复制，因此不需要给 `CHANGE MASTER` 提供任何位置参数。

不幸的是，该方法的前提假设不总是成立的，即 slave 能够接收到 master 上的所有更新。通常来说，slave 可能在不同程度上落后于 master。可能只是几个事务而已，但不管怎样，slave 始终还是落后了。也就是说，每个 slave 都需要获取丢失的事务，而如果其他 slave 都没有启用二进制日志，这就无法实现。解决办法是，找到获取最多 master 变更（即知道最多）的 slave，然后将其他 slave 与这个知道最多的 slave 同步：复制整个数据库，或者使用类似 *mysqldbcompare* 的工具获取变更。

不管怎样，这种方法很简单。如果你要处理丢失的事务，或者运行负载较低，那么这个方法很有用。

## 提升 slave 的修正方法

147

大多数情况下，提升 slave 的传统方法并不适合，因为 slave 往往落后于 master。图 5-8 说明了 master 意外故障的典型情况。其中，中间写有“二进制日志”的方框是 master 的二进制日志，箭头代表 slave 执行了多少二进制日志。

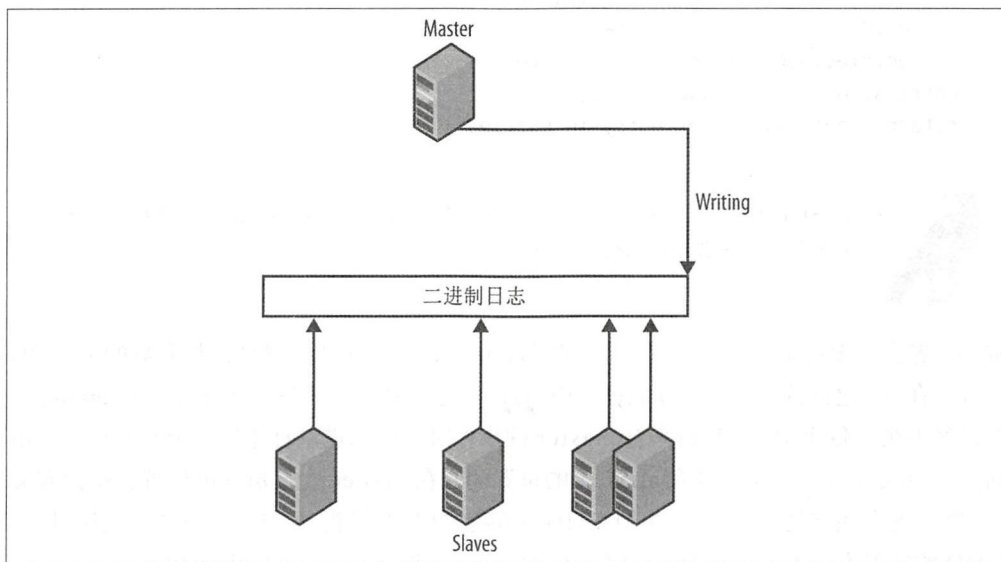


图5-8: master及其连接的slave的二进制日志位置



图中，每个 slave 在不同的 binlog 位置停止，即便是知道最多的那个 slave，也没有从故障 master 那里获得全部变更。没有复制到新 master 的事务将会永远丢失(将在第 6 章的“级联部署的一致性”一节中解释)，落后的 slave 丢失的事务需要从知道最多的那个 slave 那里获得。这时的解决办法是，将某个 slave（即知道最多的那个）提升为新 master，然后同步其他 slave。示例 5-4 给出了根据 master 位置对 slave 进行排序的代码，这里假定 slave 连接的是同一个 master，即位置是可比较的。

示例5-4：确定最佳slave的Python代码

```
from mysql.replicant.commands import (
    fetch_slave_position,
)

def fetch_gtid_executed(server):
    server.connect()
    result = server.sql(
        "SELECT server_id, trans_id FROM Last_Exec_Trans"
    )
    server.disconnect()
    return result

def order_slaves_on_position(slaves):
    entries = []
    for slave in slaves:
        pos = fetch_slave_position(slave)
        gtid = fetch_gtid_executed(slave)
        entries.append((pos, gtid, slave))
    entries.sort(key=lambda x: x[0])
    return [ entry[1:2] for entry in entries ]
```



MySQL 5.6 引入了 GTID，就不存在这个问题。关于 GTID 的具体介绍，参见第 8 章“全局事务标识符”一节。

重要问题在于要把每个 slave 的位置（即故障 master 上的位置）翻译成提升后的 slave 的位置。在 5.6 之前的版本中，复制过程中 slave 上已经执行的那些历史事件及其 binlog 位置已经丢失。每次 slave 执行来自 master 的事件时，向二进制日志写一个新的事件，用新的 binlog 位置。这个事件在 slave 上的位置同它在 master 上的 binlog 位置完全没有关系。唯一的办法就是，实现一个类似 GTID 的功能，然后扫描提升后的 slave 的二进制日志。GTID 的实现方法参见附录 B。示例 5-5 给出了实现提升 slave 的 Python 代码。





示例5-5：用Python实现slave提升

```
def promote_best_slave(slaves):  
    entries = order_slaves_on_position(slaves) ❶  
    _, master = entries.pop() ❷  
    for gtid, slave in entries:  
        pos_on_master = find_position_from_gtid(master, gtid) ❸  
        switch_to_master(master, slave, pos_on_master) ❹
```

- ❶ 使用附录 B 中的函数获取每个 slave 的位置，该函数通过 SHOW SLAVE STATUS 获取最后一个事件的位置。
- ❷ 选择位置最高的 slave 提升为 master。如果最高位置的 slave 不止一个，随便选一个即可。
- ❸ 连接提升后的 slave，扫描二进制日志，确定每个 slave 上最后一个事务的 GTID 位置。每个 GTID 位置对应一个提升后的 slave 上的 binlog 位置。
- ❹ 将每个 slave 重新连接到提升后的 slave，从新 master 的二进制日志位置开始。

149

## 环形复制

了解了双主结构，你可能会想是不是有可能建立一个多 master 结构，即两个以上 master 相互复制。由于每个 slave 只能连接一个 master，因此这种结构只能以环形的方式搭建。

MySQL 5.6 之前的版本并不推荐使用这种结构，但这也是有可能的。随着 5.6 中全局事务 ID 的引入，很多不推荐环形复制的原因都失效了，其中主要原因是一旦发生故障系统将无法正确运行。

考虑到位置的原因，使用三个或三个以上服务器进行环形复制是很实用的。举一个现实中的例子，比如某个移动电话运营商的用户遍布整个欧洲。由于手机用户经常漫游，如果客户的注册点离手机实际所在位置很近，那就非常方便。因此，通过在欧洲的一些战略要地放置数据中心，就可以快速验证呼叫数据，并在本地注册新的呼叫。然后这些变更被复制到环中所有的服务器，最终所有服务器都有准确的计费信息。在这种情况下，环形复制是一个完美的设置：所有的用户数据被复制到所有站点，所有的数据中心都可以进行数据更新。

建立环形复制，如图 5-9 所示，是非常简单的。示例 5-6 所示的脚本可以自动建立环形复制，那么难点在哪里呢？每次配置的时候，你都要问问自己：“如果出了问题怎么办？”

示例5-6：建立环形复制

```
def circular_replication(server_list):  
    from mysql.replicant.commands import change_master  
    for source, target in zip(server_list, server_list[1:] + [server_list[0]]):  
        change_master(target, source)
```



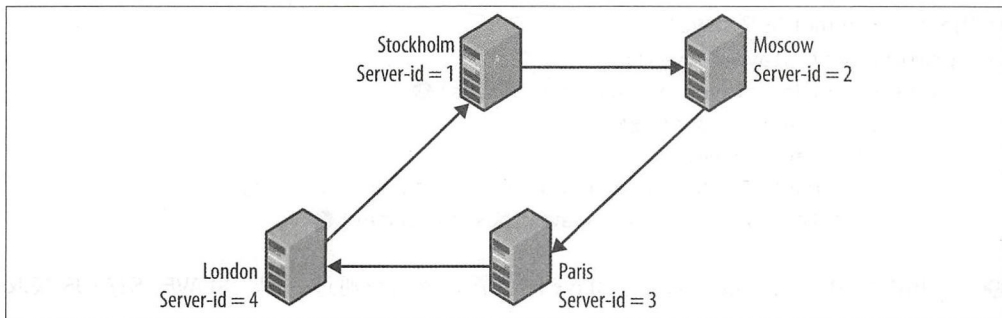


图5-9: 设置环形复制

150 图 5-9 中有 4 台以所在城市命名的服务器（名称是任意的，与配置无关）。复制以环形方式进行：从“Stockholm”到“Moscow”到“Paris”到“London”，然后再回到“Stockholm”。也就是说，“Moscow”是“Paris”的上游，是“Stockholm”的下游。假设“Moscow”突然意外停止，为了使复制继续，需要把“下游”服务器“Paris”重新连接到“上游”服务器“Stockholm”，确保系统继续运行。

图 5-10 展示了这个情景：某台服务器出现故障，其他服务器重新连接，使得复制继续。听起来很简单，对吧？其实并不像它看起来那么简单。基本来讲，有三个问题：

- 下游服务器（即连接故障 master 的 slave）需要连接上游服务器，并从最近的位置开始复制。怎么确定这个位置？
- 假设故障服务器在崩溃前已经成功地发出了一些事件。这些事件怎么处理？
- 怎样把故障服务器重新接入拓扑结构。如果服务器应用了那些已经写入二进制日志却尚未发出去的事务，会怎样？显然这些事务会丢失，所以需要处理这个问题。

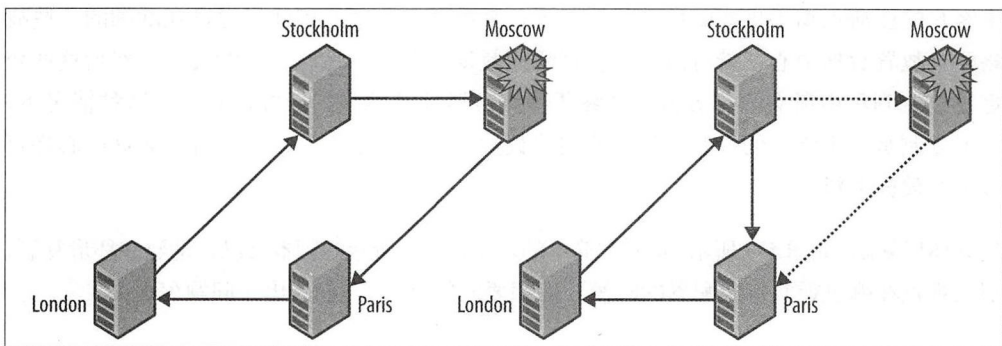


图5-10: 出现故障服务器时的拓扑变化

我们发现，所有问题都可以通过全局事务标识符（在 MySQL 5.6 中引入）轻松解决。如



果发现某个服务器出现故障，使用 `CHANGE MASTER TO` 命令加上 `MASTER_AUTO_POSITION=1` 选项，将下游服务器连接到上游服务器：

```
paris> CHANGE MASTER TO
-> MASTER_HOST='stockholm.example.com',
-> MASTER_AUTO_POSITION = 1;
```

由于每个服务器都有事务记录，故障服务器发出去的任何事务会在剩余的每个服务器上执行一次。也就是说，前面列出的第 2 个和第 3 个问题就自动解决了。

151

在未来某个时间，可能需要比较故障服务器和其他服务器（参见图 6-8），将服务器恢复到环中会导致那些丢失的事务“突然出现”（从应用程序使用数据库的角度），你应该不想这样。将服务器恢复到环中最安全的方式是，从环中任意一个服务器恢复，然后重新连接环，这样新服务器就又出现在环中了。

## 小结

在实践中，高可用性是一个重要概念。本章介绍了高可用性，以及如何在 MySQL 中实现它。下一章我们将进一步研究高可用性，讲述相关话题：横向扩展。

Joel 的邮件提示声响了。他点开邮件，打开最近一条消息。是 Summerson 先生发来的，对他的报告做了评论。通读完毕后，他在邮件末尾发现了期望看到的话：“我喜欢这个冗余的想法，特别是热备份策略。就这么做吧。”

Joel 叹了口气，意识到熟悉同事这件事必须得缓一缓了。他有太多工作要做。



# 面向横向扩展的MySQL复制

Joel 站起来伸了个懒腰，想去弄点喝的。他绕过桌子，正要去休息室，正好在办公室门口遇到老板。“下午好，先生。”

“你好，Joel。我们的新应用刚刚卖出了一批许可证。营销部门的人告诉我数据库服务器负载预期可能至少增加 10 倍。”

Joel 扬了扬眉毛。上周他刚加了一台 slave 改善了负载问题，但是没有根本解决问题。

“我们需要横向扩展，Joel。”

“是的，先生，我马上去做。”

Summerson 先生笑着拍了拍 Joel 的肩膀，然后朝他的办公室走去。

Joel 在那站了一会儿，思考什么是“横向扩展”，以及如何规划。“我得再看点资料，”他一边往休息室走一边嘀咕着。

当负载开始增加（如果部署没问题，这只是个时间问题），有两种解决办法。第一种方法是购买更强大的服务器来应对增加的负载，称为纵向扩展（或向上扩展，scale up）；第二种方法是添加更多的服务器，称为横向扩展（或向外扩展，scale out）。其中，横向扩展更常用，因为它通常只需购买低成本的标准服务器，更具有成本效益。

添加的服务器不仅可以处理增加的负载，还可以支持高可用性及其他商业需求。如果有效使用，横向扩展可以综合所有的服务器的资源，比如计算能力，最大程度地利用这些资源。

关于横向扩展，本章不打算深入硬件、网络及其他相关问题，这些超出了本书的范围，可以参考“高性能 MySQL”（*High Performance MySQL*）一书。我们要讨论的是如何利



用横向扩展建立 MySQL 复制。首先介绍复制的基础知识，然后我们开发一个 Python 库来简化大量服务器的复制管理，接着讨论复制如何适应组织的业务需求。

本章（包括本书其他章节）所有代码都可以在 Launchpad (<http://bit.ly/mysqllaunch>) 上的 MySQL Replicant 源代码库中找到。

横向扩展和复制的常见用途有如下几项。

#### 读操作的负载均衡

由于 master 忙于更新数据，所以将响应查询的服务器分离出来是明智的。查询只需要读取数据，使用复制机制将 master 上的变更发给 slave（需要多少就发多少），这样 slave 上就有了当前数据，能够处理查询。

#### 写操作的负载均衡

高流量的部署将处理分发到很多计算机上，有时候有上千个。因此，复制在分发信息的过程中起着关键作用。根据数据的业务用途及使用性质，信息的分发方式有：

- 基于信息角色的分发。很少更新的表放在一个服务器上，而频繁更新的表则分割到多个服务器上。
- 按地理区域分割，这样流量可以直接定向到最近的服务器。

#### 通过热备份进行灾难避免

如果 master 出现故障，一切都会停止，就无法执行事务（可能是关键事务）、获取客户信息，也无法检索其他关键数据。这会严重影响业务，所以要（几乎）不惜任何代价避免这种情况发生。最简单的方法就是配置一个专门的 slave 作为热备份，如果 master 发生故障，随时接手 master 的工作。

#### 通过远程复制进行灾难避免

由于灾难的存在，比如断电、地震或洪水等，每个部署都有数据中心发生故障的风险。要缓解这种情况，可以在地理上的远程站点之间利用复制进行信息传输。

#### 制作备份

留一个服务器专门做备份是很常见的。这样可以在完全不影响 master 的前提下执行备份，因为将备份服务器离线，然后做任何事情。

#### 生成报表

从服务器上的数据创建报表会降低服务器的性能，有时候影响还很大。如果生成报表需要大量后台作业，那最好建立一个专门的 slave 来完成这项工作。要想获得数据库某个时间点上的快照，先停止 slave 上的复制，然后在不影响主业务服务器的

情况下运行大量查询。例如，在某天的最后一个事务完成后停止复制，就可以在其他业务正常运行的情况下获取当天的日报表。

#### 过滤或分区数据

如果网络连接很慢，或者有些数据对某些客户端不可用，可以添加一个服务器进行数据过滤。同样适用于将数据分区存到独立的服务器。

## 横向扩展读操作，而不是写操作

理解横向扩展方法只能扩展读操作而不是写操作，这点很重要。每个新 slave 处理的写负载与 master 一样。系统的平均负载为：

$$AverageLoad = \frac{\sum ReadLoad + \sum WriteLoad}{\sum Capacity}$$

如果单个服务器每秒有 10 000 个事务，master 每秒的写负载为 4 000 个事务，而每秒的读负载为 6 000 个事务，结果就是：

$$AverageLoad = \frac{\sum ReadLoad + \sum WriteLoad}{\sum Capacity} = \frac{6000 + 4000}{10000} = 100 \%$$

现在，如果添加 3 个 slave，每秒的总负载量就增加到 40 000 个事务。由于写操作也会被复制，每个写操作都会执行 4 次（1 次在 master 上，3 个 slave 各执行 1 次），每个 slave 的写负载仍然是每秒 4 000 个事务。而由于读负载被分发到各个 slave，所以总的读负载并没有增加。那么，现在的平均负载就是：

$$\boxed{156} \triangleright AverageLoad = \frac{\sum ReadLoad + \sum WriteLoad}{\sum Capacity} = \frac{6000 + 4 * 4000}{4 * 10000} = 55 \%$$

注意，公式中因为有 4 台服务器，所以总负载事务量增至 4 倍，同时复制导致写负载也增至 4 倍。

经常容易忘记的是，master 的所有写查询都会被复制到各个 slave。所以用这种简单的方法并不能扩展写操作，而是扩展读操作。后面我们将介绍一种称为分片（sharding）的技术扩展写操作。

# 异步复制的价值

MySQL 复制是异步（asynchronous）的，这种复制特别适合现代应用，如网站等。

为了处理大量的读请求，各个站点通过复制创建 master 的副本，然后 slave 处理所有的读请求，master 处理写请求。这个复制过程是异步的，因为 master 不需要等待 slave 应用变更，而是将每个变更请求分发到各个 slave，假定它们最终会达到一致性且应用了所有变更。在横向扩展的时候，这种技术能够很好地提高性能。

相反，同步复制要求 master 和 slave 保持同步，如果 slave 不同意提交事务，master 也不能提交这个事务。也就是说，同步复制要求 master 必须等待所有 slave 的写操作完成。

异步复制比同步复制快得多，很快你就会明白为什么。与异步复制不同，同步复制需要额外的同步机制来保证一致性，一般通过两阶段提交（two-phase commit）协议来实现。两阶段提交协议保证了 master 和 slave 之间的一致性，但却需要它们之间有额外的通信消息传递。一般来说，工作过程如下：

1. 当执行提交语句时，事务被发送到 slave，slave 开始准备事务的提交。
2. 每个 slave 都要准备事务，然后向 master 发送 OK（或 ABORT）消息，表明事务已经准备好（或者无法准备）。
3. master 等待所有 slave 发来 OK 或 ABORT 消息：
  - a. 如果 master 收到所有 slave 的 OK 消息，它就会向所有 slave 发送提交消息，告诉它们提交该事务。
  - b. 如果 master 收到任何一个 slave 的 ABORT 消息，它就向所有 slave 发送 ABORT 消息，告诉它们中止事务。
4. 然后每个 slave 等待 master 发送 OK 或 ABORT 消息。
  - a. 如果 slave 收到提交请求，它们就会提交事务，并向 master 发送事务已提交的确认。
  - b. 如果 slave 收到中止请求，它们就会撤销所有变更并释放资源，从而中止事务，然后向 master 发送事务已中止的确认。
5. 当 master 收到所有 slave 的确认后，就会报告该事务被提交（或中止），然后继续处理下一个事务。

这个协议之所以慢，是因为它一共需要 4 次消息传递，包括事务消息和准备请求的消息。主要问题不在于处理同步的网络流量，而是由于网络和 slave 提交产生的延迟，而且 master 的提交会被阻塞直到所有 slave 确认事务。而异步复制只需要一条事务消息即可，这样的好处是，master 不需要等 slave，就可以立即报告事务的提交，从而极大地提高了性能。

那么，在同步复制中，slave 处理事务时为什么会阻塞提交呢？如果 slave 离 master 很近，同步复制所需的额外消息传递就不会有什么影响，但是如果 slave 不近——可能在另一个城镇甚至另一个洲——那就大不相同了。

表 6-1 列举了一些例子，其中服务器每秒能够提交 10 000 个事务，即提交时间为 0.1ms（请注意，有些实现可以并行处理多个独立事务的提交，比如 MySQL 集群）。如果网络延迟为 0.01ms（这个基值通过 ping 本地电脑而得），事务提交时间就会增加为 0.14ms，即每秒约提交 7 000 个事务。如果网络延迟为 10ms（通过 ping 附近城市的服务器而得），事务提交时间就会增加到 40.1ms，即每秒约提交 25 个事务！相反，异步复制完全没有延迟，事务被立即提交，所以事务提交时间保持不变，仍然是最初的每秒 10 000 个事务，就好像没有 slave 一样。

158

表 6-1：同步复制引起性能下降的典型例子

延迟 (ms)	事务提交时间 (ms)	每秒提交的事务数量	示例
0.01	0.14	~7100	Same computer
0.1	0.5	~2000	Small LAN
1	4.1	~240	Bigger LAN
10	40.1	~25	Metropolitan network
100	400.1	~2	Satellite

异步复制其实是以牺牲一致性为代价换取性能。回想一下，异步复制中事务的提交是立即报告的，不用等待任何来自 slave 的确认。也就是说，master 认为已经提交的事务，可能还没在 slave 上提交。实际上，可能这个事务甚至还没离开 master，还在等待被发送到 slave。

有两个问题需要注意：

- 如果 master 出现故障，事务就会“消失”。
- slave 上执行的查询可能会返回旧数据。

稍后我们会讨论如何保证读取的是当前数据，目前只要记住异步复制中有些问题需要我们去处理即可。

## 管理复制拓扑

通过创建新 slave，然后将它们添加到拓扑中可以扩展部署。术语复制拓扑（replication topology）是指通过复制连接服务器的方式。图 6-1 给出了复制拓扑的示例：简单拓扑、树形拓扑、双主拓扑和环形拓扑。



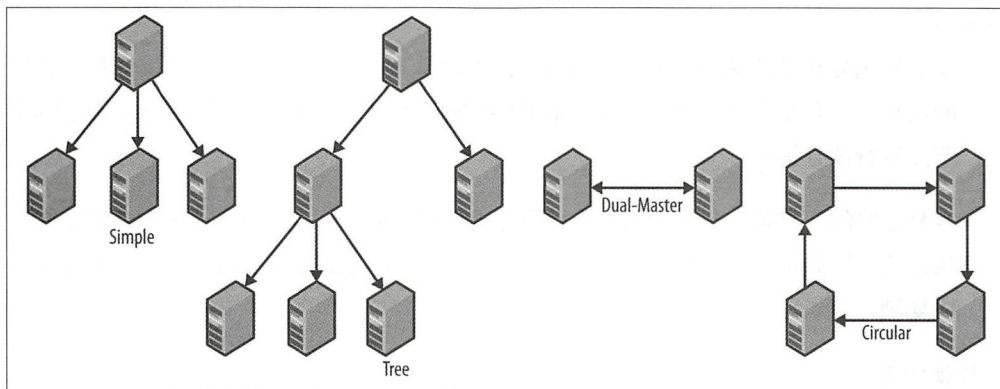


图6-1：简单拓扑、树形拓扑、双主拓扑和环形拓扑

这些拓扑用途各不相同。例如，双主拓扑用来处理故障转移，环形复制和双主结构允许各个站点在本地运行的同时还能将变更复制到其他站点。

简单拓扑和树形拓扑用于横向扩展。复制的使用导致读操作的数量远远超过了写，这种部署有两种特殊要求。

159

#### 需要负载均衡

术语负载均衡（load balancing）是指服务器之间划分查询的方式。复制是负载均衡的原因同时也是解决方案。首先，将写操作定向到 master，读操作交给 slave，复制完成对负载的基本划分。此外，有时候需要把特定的查询发给特定的 slave。

#### 需要管理拓扑

服务器早晚有崩溃的时候，这时就不得不进行替换。slave 崩溃可能并不急着替换，但是 master 崩溃必须尽快替换。

另外，如果 master 崩溃了，客户端将会被重定向到新的 master。如果 slave 崩溃，将这个 slave 移出负载均衡器池，保证不再有查询请求发给它。

为了处理负载均衡和管理，我们需要一些管理复制拓扑的工具，特别是监控服务器状态和性能的工具，以及处理分布式查询的工具。

为了使负载均衡更加高效，服务器要保留空闲处理能力，原因是：

#### 峰值负载的处理

要有余力处理峰值负载。系统负载从来不是均匀变化的，而是上下波动的。空闲处理能力很大程度上取决于应用程序，所以需要密切监控应用以确定什么时候响应时间变慢。

**160** 分布成本

要有空闲处理能力来运行复制。复制总是需要“浪费”一些处理能力在分布式系统的运行上，包括管理分布式系统所需的额外查询，比如需要额外的查询来确定哪个节点执行读查询。

容易忽略的一点是，每个 slave 执行的写操作和 master 一样。master 的查询是有序执行的（比如串行），不会产生变更冲突，而 slave 则需要一些额外的处理能力来完成复制。

**管理性任务**

重新建立复制需要空闲处理能力，以便临时做些其他的工作，例如，在服务器之间移动数据的时候。

两种情况下需要负载均衡：一种是应用程序根据查询类型请求服务器，另一种是中间层（通常指代理）分析查询，然后发给适当的服务器。

使用中间层分析和分发查询（如图 6-2 所示）是目前最灵活的方法，但有两点不足：

- 使用代理会导致性能下降，原因有两个：1) 分析查询需要消耗资源，2) 查询必须经过代理，多了一次节点转移（hop）。这会延迟查询，因为查询会被解析和分析两次：一次是代理，另一次是 MySQL 服务器。此外，多出来的节点转移导致的延迟可能比分析查询的时间还长。根据具体应用不同，这或许是个问题，也可能不是。
- 正确的查询分析很难实现，有时甚至不可能实现。代理通常向应用程序员隐藏了部署的内部结构，使应用程序不用考虑部署选择。但是正因为如此，客户端所发出的查询可能很难被正确分析，而且可能需要在发给服务器之前进行大幅度重写。

要实现基于代理的负载均衡，其中一个工具就是 MySQL Proxy，其中包含 MySQL 客户端协议的完整实现，所以它既可以作为服务器用于客户端连接，也可以作为客户端连接到 MySQL 服务器。这是完全透明的，客户端不用区分是代理还是真正的服务器。

**161** 使用 Lua 编程语言控制 MySQL 代理，其内置的 Lua 引擎通过执行一些小程序（有时候并不小），拦截和操纵查询及其结果集。因为使用真实的编程语言控制代理，所以还能执行各种复杂的任务，包括查询分析、查询过滤、查询操作和查询分发等。

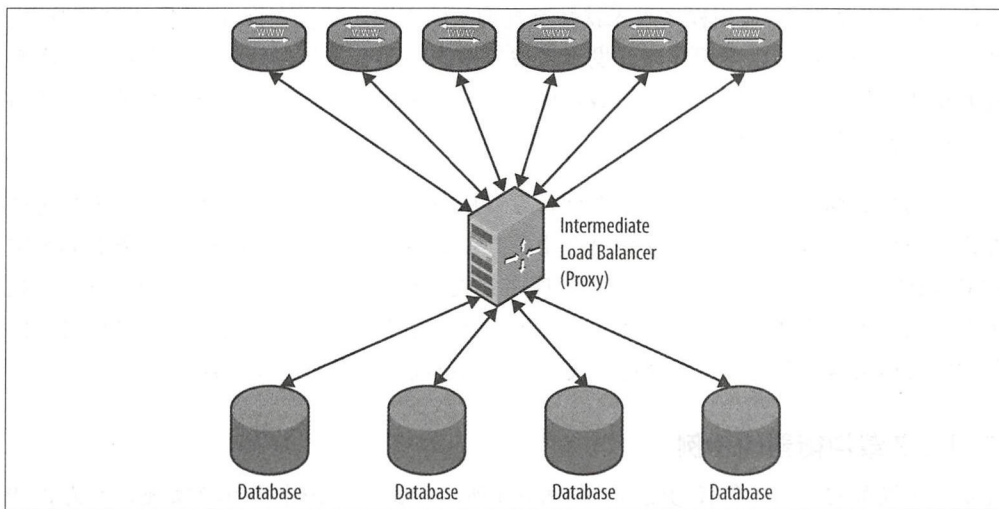


图6-2：使用代理分发查询

MySQL 代理的配置和编程超出了本书的范围。网上有很多这方面的出版物，下面是几个我们认为比较有用的：

Jan Kneschke (<http://jan.kneschke.de>)

Jan Kneschke 是 MySQL 代理的最初作者，他有很多关于代理的讲演和文章。

MySQL 参考手册 (<http://bit.ly/mysql-56-ref>)

MySQL 参考手册中的“MySQL Proxy”一节讲了很多实现细节，并介绍了如何编写 MySQL Proxy 脚本。

代理的具体使用方法完全取决于代理类型，所以这里不讨论这个问题，而是研究在应用层使用负载均衡器的问题。负载均衡器有很多，包括：

- 硬件
- 简单软件负载均衡器，如 Balance (<http://www.inlab.de/balance.html>)
- 基于对等的系统 (Peer-based systems)，如 Wackamole (<http://www.backhand.org/wackamole/>)
- 完全成熟的集群方案，如 Linux 虚拟服务器 (<http://www.linuxvirtualserver.org/>)

◀ 162

此外，还可以在 DNS 级别分发负载，或者直接在应用程序中分发负载。

## 应用层的负载均衡

应用层负载均衡最直接的办法就是：应用程序根据要发出的查询类型向负载均衡器请求

连接。大部分情况下,应用程序事先就知道查询是读查询还是写查询,以及会用到哪些表。其实,将查询设计成由应用程序开发者考虑这些问题还有其他好处的,通常可以提高系统的整体性能。根据这些信息,负载均衡器可以将应用程序连接到正确的服务器,然后应用程序就可以执行查询了。

应用层的负载均衡器需要一个中心存储,存储服务器信息以及这些服务器能够进行哪些查询。应用层的函数将查询发送到这个中心存储,返回要使用的 MySQL 服务器的名字和 IP 地址。这个查找过程可以在应用程序中完成,或者也可以在负载均衡器的连接器(如果有的话)内部完成。很多连接器可以在不需要中心存储的情况下提供服务器信息,但是它需要连接信息在连接器上登记,或者也可以通过应用程序提供服务器信息。

### 应用层负载均衡器的示例

开发一个简单的应用层负载均衡器,如图 6-3 所示。表示层逻辑用 PHP 实现,因为 PHP 广泛用于 Web 服务器。需要编写两个函数,分别用于更新服务器池信息和从池中获取服务器。

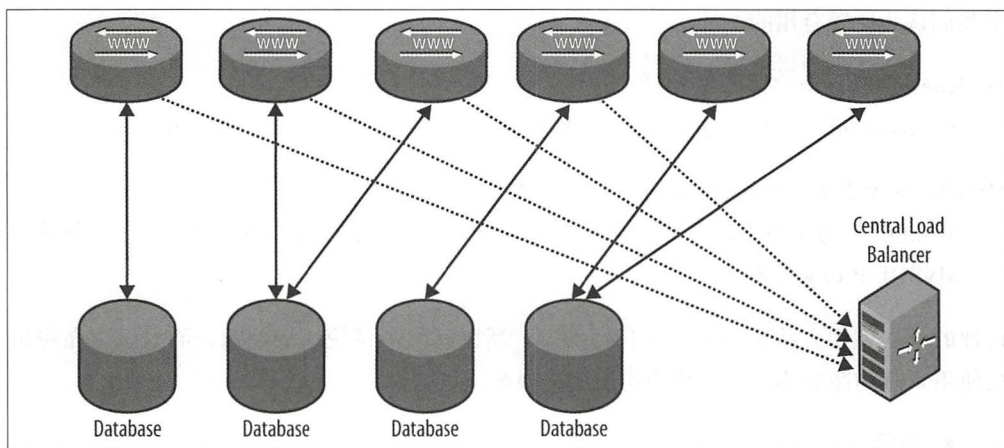


图6-3: 应用层的负载均衡

这个池是这样实现的: 在一个公共数据库上创建一个表存储所有服务器的相关信息, 这个数据库由所有节点共享。使用主机+端口作为该表的主键(而不需要创建 hostID 字段), 然后创建一个含有共享数据表的公共数据库。

163 > 应该为中心存储做个副本, 这样才不会产生单点故障。另外, 由于可用服务器列表通常不会变化, 所以负载均衡信息很适合缓存。

为了简单起见, 避免引入对其他系统的依赖性, 我们使用纯 MySQL 实现来讲解应用层



负载均衡器。你也可以使用其他非 MySQL 技术，最常用的是使用轮换调度的 DNS，或者使用分布式内存键值存储 Memcached。

还要注意，增加额外查询会给高性能系统带来重大开销，应该避免。通常可以通过缓存实现，不过这里我们先考虑没有缓存的情况。如何添加缓存参见示例 6-5。

负载均衡器将服务器目录放在负载均衡器池中，根据它们所处理的查询类型分成不同类别。池中的服务器信息存在一个中心存储中。具体实现包括：公共数据库表（参见示例 6-1），供应用程序查询的负载均衡器（参见示例 6-2），以及更新服务器信息的 Python 函数（参见示例 6-3）。

示例6-1：负载均衡器的数据库表

```
CREATE TABLE nodes (
  host VARCHAR(28) NOT NULL,
  port INT UNSIGNED NOT NULL,
  sock VARCHAR(80)
  type ENUM('OL','RO','RW') NOT NULL DEFAULT '',
  PRIMARY KEY (host, port)
);
```

164

这个存储包含服务器的主机和端口信息，以及它是否是离线（OL）、只读（RO）或可读写（RW）的服务器。离线设置用于维护。

示例 6-2 列出了维护负载均衡器的代码，其中有一个字典类，负责向服务器分配连接。

示例6-2：负载均衡器的PHP代码

```
define('DB_OFFLINE', 'OL')
define('DB_RW', 'RW');
define('DB_RO', 'RO');

$FETCH_QUERY = <<<END_OF_QUERY
SELECT host, port FROM nodes ❶
  WHERE FIND_IN_SET(?, type)
ORDER BY RAND() LIMIT 1
END_OF_QUERY;

class Dictionary { ❷
  private $server;

  public function __construct($host, $user, $pass, $port = 3306) {
    $this->server = new mysqli($host, $user, $pass, 'metainfo', $port);
  }

  public function get_connection($user, $pass, $db, $hint) { ❸
```

```

global $FETCH_QUERY;
$type = $hint['type'];
if ($stmt = $this->server->prepare($FETCH_QUERY)){
    $stmt->bind_param('s', $type);
    $stmt->execute();
    $stmt->bind_result($host, $port);
    if ($stmt->fetch())
        return new mysqli($host, $user, $pass, $db, $port);
}
return null;
}
}

```

- ❶ 简单的 SELECT 语句可以找到所有接受查询的服务器。由于我们只需要一个服务器，在 SELECT 语句后面加上 LIMIT 修饰符，将输出限制为一行，然后使用 ORDER BY RAND() 修饰符，将查询平均地分散到所有可用的服务器上。
- ❷ 使用了字典类（后面还会用到），负责将连接分配给 MySQL 实例。创建字典类实例的时候需要提供存储相关信息的 MySQL 服务器。为了管理连接，这个服务器存储了部署中每个服务器的信息。
- ❸ get\_connection 函数用于部署中服务器请求连接。根据 get\_connection 的 hint 参数确定服务器将要连接的对象。hint 是一个关联数组，表示请求何种连接，然后函数会根据这个标准返回一个连接。这时，hint 只表示请求到的服务器是只读的还是可读写的。

最后，需要提供添加和删除服务器及更新服务器的工具函数。由于这些任务主要供管理逻辑使用，因此利用 Replicant Python 库来实现。这个工具由以下三个函数组成。

`pool_add(common, server, type)`

向池中添加服务器 *server*。池存储在 *common* 服务器上，*type* 是列表类型（或其他可迭代类型），表示需要设置的值。

`pool_del(common, server)`

从池中删除服务器。

`pool_set(common, server, type)`

改变服务器 *server* 的类型。

示例6-3：负载均衡器的管理性函数

```

from mysql.replicant.errors import (
    Error,
)

```

```

from MySQLdb import IntegrityError

class AlreadyInPoolError(Error):
    pass

_INSERT_SERVER = ("INSERT INTO nodes(host, port, sock, type)"
                  "VALUES (%s, %s, %s, %s)")

_DELETE_SERVER = ("DELETE FROM nodes"
                  " WHERE host = %s AND port = %s")

_UPDATE_SERVER = ("UPDATE nodes SET type = %s"
                  " WHERE host = %s AND port = %s")
def pool_add(common, server, types=None):
    if types is None:
        types = []
    common.use("common")
    try:
        common.sql(_INSERT_SERVER,
                   (server.host, server.port, server.socket, ','.join(types)))
    except IntegrityError:
        raise AlreadyInPoolError

def pool_del(common, server):
    common.use("common")
    common.sql(_DELETE_SERVER,
               (server.host, server.port))

def pool_set(common, server, types=None):
    if types is None:
        types = []
    common.use("common")
    common.sql(_UPDATE_SERVER,
               (','.join(types), server.host, server.port))

```

166

这些函数的使用方法如以下示例所示：

```

pool_add(common, master, ['READ', 'WRITE'])

for slave in slaves:
    pool_add(common, slave, ['READ'])

```

一切就绪以后，就可以像示例 6-4 那样使用负载均衡器了。将字典设置为使用 `central.example.com` 作为中心存储，然后 `get_connection` 根据 `hint` 获取服务器连接。

示例6-4: 使用负载均衡器的PHP代码

```
$DICT = new Dictionary("central.example.com", "mats", "");

$QUERY = <<<END_OF_QUERY
SELECT first_name, last_name, dept_name
FROM employees JOIN dept_emp USING (emp_no)
        JOIN departments USING (dept_no)
WHERE emp_no = ?
END_OF_QUERY;

$mysql = $DICT->get_connection('mats', 'xyzyz', 'employees',
                                array('type' => DB_RO));
$stmt = $mysql->prepare($QUERY);
if ($stmt) {
    $stmt->bind_param("d", $emp_no);
    $stmt->execute();
    $stmt->bind_result($first_name, $last_name, $dept_name);
    while ($stmt->fetch())
        print "$first_name $last_name $dept_name\n";
    $stmt->close();
}
else {
    echo "Error: " . $mysql->error;
}
```

167

在示例 6-2 中, 查询被发送到中心存储以供分配。这就使应用程序发出的查询加倍, 会导致性能下降。为此, 应该将中心存储的数据缓存起来, 然后直接从缓存里取数据, 如示例 6-5 所示。

需要确定什么时候缓存失效。这个例子中, 使用简单的生存时间 (time to live) 策略, 即如果缓存太旧则重新加载。这是一个很简单的实现方式, 但这时无法立即识别任何拓扑变更。如果拓扑变化了, 改变了中心存储的信息, 必须要等生存时间过了才能删除旧的服务器, 然后从中心存储重新加载信息。

示例6-5: 缓存负载均衡器的PHP代码

```
define('DB_RW', 'RW');
define('DB_RO', 'RO');
define('TTL', 60); ❶

$FETCH_QUERY = <<<END_OF_QUERY
SELECT host, port, type FROM nodes ❷
END_OF_QUERY;
```



```

class Dictionary {
    private $server;
    private $last_update;
    private $cache;

    public function __construct($host, $user, $pass, $port = 3306)
    {
        $this->server = new mysqli($host, $user, $pass,
                                   'metainfo', $port);
    }

    public function get_connection($user, $pass, $db, $hint)
    {
        if (time() > $this->last_update + TTL) ❸
            $this->update_cache();
        $type = $hint['type'];
        if (array_key_exists($type, $this->cache)) {
            $servers = $this->cache[$type];
            $no = rand(0, count($servers) - 1);
            list($host, $port) = $servers[$no]; ❹
            return new mysqli($host, $user, $pass, $db, $port);
        }
        else
            return null;
    }

    private function update_cache() {
        global $FETCH_QUERY;
        if ($stmt = $this->server->prepare($FETCH_QUERY)){ ❺
            $cache = array();
            $stmt->execute();
            $stmt->bind_result($host, $port, $type);
            while ($stmt->fetch()) ❻
                $cache[$type][] = array($host, $port);
            $this->cache = $cache;
            $this->last_update = time();
        }
    }
}

```

168

- ❶ 这个常量设置缓存的生存时间。如果这个时间很长，说明不经常查询中心存储，但也意味着拓扑变更没那么快被发现。
- ❷ 与示例 6-2 不同，中心存储的全部内容都会加载到查询中。这里假定加载全部内容，但对大型数据集来说，如果查询不加载字典表中不会用到的信息，这是个更明智的选择。

- ③ 检查上次更新缓存的时间。如果这个时间超过了 TTL，则更新缓存。if 语句执行完毕后，缓存是最新的（或者至少足够新）。
- ④ 从缓存而不是服务器获取主机和端口，如示例 6-2 那样。这里，服务器是任意选择的，但也可以采用其他方式。
- ⑤ 如果能够在服务器上准备查询，则只更新缓存。如果由于某种原因无法联系服务器，还是需要执行查询。在这段代码中，假定数据库重启时，至少有那么一会儿使用的是缓存的当前内容。
- ⑥ 这里基于服务器类型填充缓存。缓存中的每一条记录是某种类型的可用服务器列表。

## MySQL 原生驱动器的复制和负载均衡插件

MySQL 的 PHP 小组创建了几个 MySQL 原生驱动器 (*mysqlnd*) 的插件。其中有一个插件能够用来处理读写分离、负载均衡（有几种不同策略）和故障转移。更多内容请看 PHP.net (<http://bit.ly/mysqlnd>)。

169 与前面的实现不同，*mysqlnd\_ms* 有一个配置文件，存储往哪里转移故障的信息。这很高效（所有信息都在内存中），但这是静态的。

配置文件中以 JSON 格式存储了 master 和 slave 的相关信息，类似示例 6-6。这里假定 master 是可读写的，而 slave 是只读的。

示例 6-6: *mysqlnd\_ms* 配置文件示例

```
{
  "myapp": {
    "master": [
      { "host": "master1.example.com" }
    ],
    "slave": [
      { "host": "slave1.example.com", "port": "3306" },
      { "host": "slave2.example.com", "port": "3307" },
      { "host": "slave3.example.com", "port": "3308" }
    ]
  }
}
```

在建立连接的时候，将主机名作为示例 6-6 结构中的关键字，如果找到匹配的值，则直接使用那条记录的连接信息。使用哪个连接信息取决于负载均衡器的策略集。负载均衡器审查语句，然后决定将语句发给谁。任何以 SELECT 开头的语句都是只读语句，将发给 slave，而其他所有语句都发给 master。使用 *mysqlnd\_ms* 的代码如示例 6-7 所示。

示例6-7: 使用mysqlnd\_ms的PHP代码

```
$QUERY = <<<END_OF_QUERY
SELECT first_name, last_name, dept_name ❶
FROM employees JOIN dept_emp USING (emp_no)
           JOIN departments USING (dept_no)
WHERE emp_no = ?
END_OF_QUERY;

$mysqli = new mysqli("myapp", "mats", "xyzyzy", "employees"); ❷
$stmt = $mysqli->prepare($QUERY);
if ($stmt) {
    $stmt->bind_param("d", $emp_no);
    $stmt->execute();
    $stmt->bind_result($first_name, $last_name, $dept_name);
    while ($stmt->fetch())
        print "$first_name $last_name $dept_name\n";
    $stmt->close();
}
else {
    echo "Error: " . $mysqli->error;
}
```

170

- ❶ 查询包含 SELECT, 所以插件认为这是个只读查询, 应该发给 slave。
- ❷ 注意, 这里给出的主机名不是真正的主机名, 而是对配置文件中 myapp 键的引用。插件根据这个信息将查询分配到正确的服务器。

## 级联复制

尽管 master 可以很好地处理大量的 slave, 但在负载均衡器超出负荷之前 (从实践上说每个 master 最多能处理 70 个 slave, 但其实这很大程度上取决于应用程序), master 无响应永远是个问题。这时, 需要添加一个 (或多个) 额外的 slave 作为中继 slave (或简称中继服务器, relay), 其目的是通过管理一群 slave 来减轻 master 上的复制负载。这种使用中继的方式称为级联复制 (hierarchical replication)。图 6-4 描述了一个典型设置, 包括一个 master、一个 relay 和若干连接到 relay 的 slave。

默认情况下, slave 从 master 那里得到的变更不会写入 slave 的二进制日志中, 所以如果按照之前的设置在 slave 上执行 SHOW BINLOG EVENTS 命令, 将看不到 binlog 有任何事件。这是因为浪费磁盘空间记录变更是没有意义的: 如果出现问题, 比如说 slave 崩溃, 总是可以通过克隆 master 或另一个 slave 来恢复它。

171

而另一方面, relay 需要二进制日志记录所有变更, 因为 relay 需要把变更传给其他 slave。与普通 slave 不同的是, relay 本身并不需要应用这些变更, 因为它不响应任何查询。

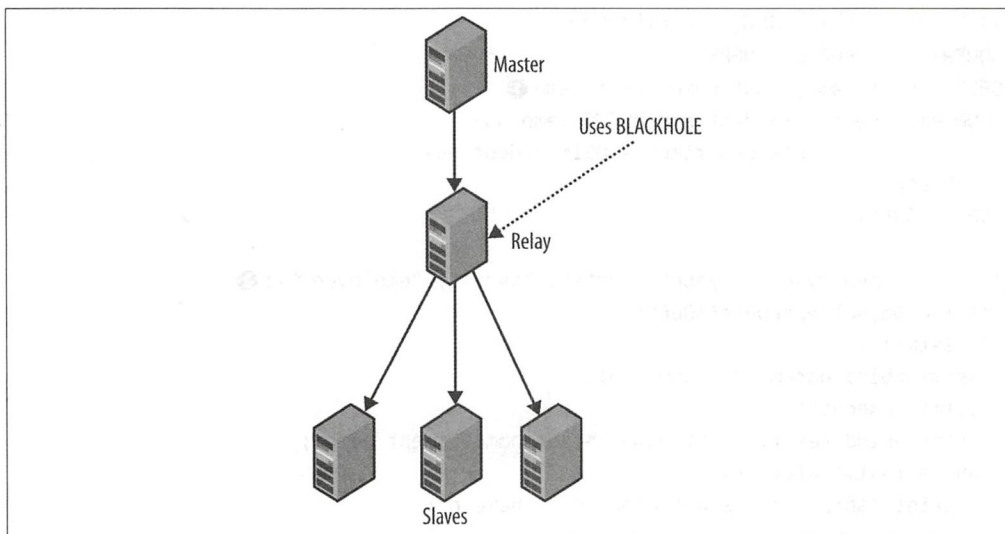


图6-4: master、relay和slave构成的级联拓扑

简而言之，普通 slave 需要将变更应用到数据库中，但不需要写二进制日志；relay 需要写二进制日志，但不应用变更。

为了避免更改数据库中的数据，需要使用表（从而语句才能执行），但丢弃其变更。为此，建立一个称为 Blackhole 的存储引擎，它接受所有语句，总是报告这些语句执行成功，但丢弃任何数据变更。reply 引入了额外延迟，这样 slave 滞后 master 的程度比 slave 与直接连接 master 的时候还多。要权衡这种滞后与 master 减少的负载量，因为管理级联设置比简单设置困难得多。

## 配置 relay

配置 relay 相当容易，但我们需要考虑将服务器角色改成 relay 的时候，怎么处理 relay 上创建的表和 relay 上已经存在的表。不在数据库中保存数据可以加快处理，而且减少了复制后的 slave 滞后，因为没有数据更新。要设置一个 relay，需要：

1. 将 slave 配置成发送任何 slave 线程执行的事件，并将这些事件写入 relay 的 binlog。
2. 将 relay 上所有表的存储引擎都改成 BLACKHOLE 存储引擎，保留空间并提高性能。
3. 保证 relay 上的所有新表都使用 BLACKHOLE 引擎。

前面提过，通过向 *my.cnf* 文件中添加 *log-slave-updates* 选项，配置 relay 发送 slave 线程事件。

除了 *log-slave-updates* 设置，还要向 *my.cnf* 文件中添加 *default-storage-engine* 更



改默认的存储引擎。通过命令 `SET STORAGE_ENGINE = 'BLACKHOLE'` 可以暂时修改 relay 的存储引擎，但服务器重启无效。

最后一步是将 relay 上已有的表的存储引擎更改为 BLACKHOLE。使用 ALTER TABLE 语句将每个表的存储引擎改成 BLACKHOLE。由于 ALTER TABLE 语句不应该被写入二进制日志（我们当然不希望 slave 丢弃它收到的变更），所以执行 ALTER TABLE 语句时要临时关闭二进制日志，参见示例 6-8。

示例6-8：更改数据库windy中所有表的存储引擎

```
relay> SHOW TABLES FROM windy;
+-----+
| Tables_in_windy |
+-----+
| user_data       |
.
.
.
| profile         |
+-----+
45 row in set (0.15 sec)
relay> SET SQL_LOG_BIN = 0;
relay> ALTER TABLE user_data ENGINE = 'BLACKHOLE';
.
.
.
relay> ALTER TABLE profile ENGINE = 'BLACKHOLE';
relay> SET SQL_BIN_LOG = 1;
```

将服务器变成 relay 要做的就是这些。通常一开始所有 slave 都直接连接到 master，一段时间之后发现需要引入 relay。原因通常是 master 负载过重，当然也可能是因为某些架构需要。那么，该怎么做呢？

可以使用前面章节所学的知识，更改已有的部署，引入新的 relay，方法是：

1. 将 relay 连接到 master，并将其角色配置为 relay。
2. 依次将 slave 的连接切换到 relay。

## 使用 Python 添加 relay

现在我们考虑扩展 Replicant 库，增加对管理 relay 的支持。我们已经有创建角色及将角色 imbue 到服务器的方法，现在我们用它们来为 relay 服务器定义一个特殊的角色，如示例 6-9 所示。

```
from mysql.replicant import roles
```

```
class Relay(roles.Role):
    def __init__(self, master):
        super(Relay, self).__init__()
        self.__master = master

    def imbue(self, server):
        config = server.get_config()
        self._set_server_id(server, config)
        self._enable_binlog(server)
        config.set('mysqld', 'log-slave-updates' '1')
        server.put_config(config)
        server.sql("SET SQL_LOG_BIN = 0")
        for db in list of databases:
            for table in server.sql("SHOW TABLES FROM %s", (db)):
                server.sql("ALTER TABLE %s.%s ENGINE=BLACKHOLE",
                           (db,table))
        server.sql("SET SQL_LOG_BIN = 1")
```

## 专用 slave

在简单的横向扩展部署中（就像我们描述的例子），所有 slave 都接收所有数据，因此能够处理任何类型的查询。但是，将请求平均分配在不同数据上并不常见。通常，有些数据需要频繁访问，而有些则很少被访问。例如，对一个电子商务网站来说：

- 产品目录几乎总被浏览。
- 可能不会经常访问产品库存数据。
- 用户数据不太经常被访问，因为大部分重要信息是以浏览器 cookie 的形式存储的会话相关信息。
- 另一方面，如果禁用 cookie，那么几乎每个页面请求都会从服务器请求会话数据。
- 新添加的项目通常比旧的访问更频繁，例如，“特价优惠”可能比其他项访问得更频繁。

显然，将那些很少访问的数据保存在每一个 slave 上是一种资源浪费。按照图 6-5 所示的那样部署更好，即几个服务器专门存储很少访问的数据，另外一组服务器专门存储频繁访问的数据。

为此，需要在复制的时候分离表。MySQL 通过过滤事件实现，在事件离开 master 或到达 slave 的时候过滤它们。

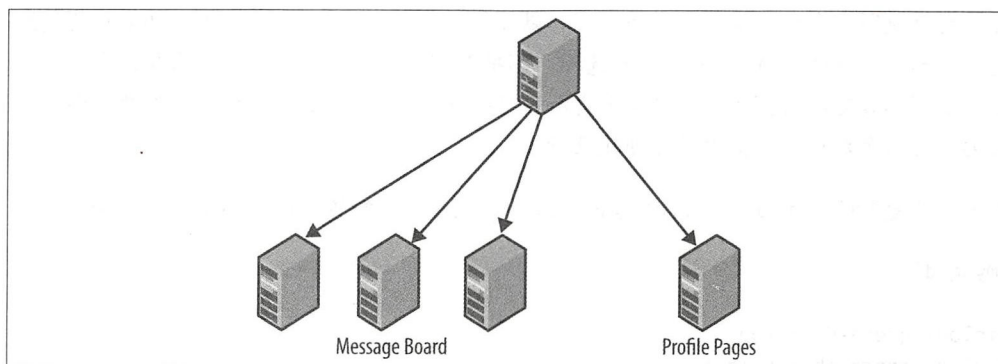


图6-5: master和专用slave的复制拓扑

## 过滤复制事件

过滤事件有两种方法：1) 在 master 上过滤事件，称为 master 过滤器；以及 2) 在 slave 上过滤事件，称为 slave 过滤器。master 过滤器控制哪些被写入二进制日志，以及哪些会发送给 slave，而 slave 过滤器控制哪些在 slave 上执行。对 master 过滤器来说，已经被过滤的表事件不会写入二进制日志。对 slave 过滤器来说，事件会写入二进制日志然后发送到 slave，直到事件执行时才进行过滤。

这意味着无法使用 PITR 正确地恢复数据库——如果数据库以备份映像的形式存储，恢复备份时是可以恢复的，但在那之后的任何数据库表的变更都无法恢复，因为二进制日志中没有记录这些变更。

如果使用 slave 过滤器，所有变更都会通过网络传送。

显然这样会浪费带宽，特别是远程网络连接。

本章后面将详细讨论 master 和 slave 过滤的优点，以及保证二进制日志完整又能节省网络带宽的方法。

### master 过滤器

创建 master 过滤器需要配置两个选项：

`binlog-do-db=db`

如果当前数据库是 `db`，则会将该语句写入二进制日志，否则忽略。

`binlog-ignore-db=db`

如果当前数据库是 `db`，则忽略该语句，否则写入二进制日志。

如果想复制所有东西，除了几个数据库，就使用 `binlog-ignore-db` 选项；如果只想复制几个数据库，就使用 `binlog-do-db` 选项。不推荐同时使用两个选项，因为逻辑上很难判断到底要不要复制数据库（如图 4-3 所示）。这两个选项不接受多个数据库参数，所以如果要设置多个数据库，需要重复使用选项。

例如，要复制除了 `top` 和 `secret` 数据库以外的东西，在配置文件中添加以下选项：

```
[mysqld]
...
binlog-ignore-db = top
binlog-ignore-db = secret
```



使用 `binlog-*-db` 选项过滤事件意味着这两个数据库都不会写入二进制日志中，因而崩溃发生时也无法使用即时恢复（PITR）进行恢复。因此，如果只是过滤复制流，强烈推荐使用 `slave` 过滤器而不是 `master` 过滤器。只有当数据是易变的（volatile）并且承担得起数据丢失时，才使用 `master` 过滤器。

## slave 过滤器

`slave` 过滤有更多选项。不仅可以基于数据库过滤事件，`slave` 过滤器还可以过滤单个表，甚至使用通配符过滤一组表。

在下面的过滤规则中，`replicate-wild` 使用数据表的全名，包括数据库名和表名。这与 `LIKE` 字符串比较函数模式相同，即下画线（`_`）匹配单个字符，百分号（`%`）匹配任意长度的字符串。注意，匹配模式有一个合法周期。也就是说，数据库名和表名是分开匹配的，所以单个通配符只能应用到数据库名或表名上。

`replicate-do-db=db`

如果当前数据库是 `db`，则执行这个语句。

176 `replicate-ignore-db=db`

如果当前数据库是 `db`，则丢弃这个语句。

`replicate-do-table=db_name.tbl_name`

`replicate-wild-do-table=db_pattern.tbl_pattern`

如果正在更新的表名为 `db_name.tbl_name`，或者表名匹配 `db_pattern.tbl_pattern` 模式，则执行表的更新。



```
replicate-ignore-table=db_name.tbl_name
```

```
replicate-wild-ignore-table=db_pattern.tbl_pattern
```

如果正在更新的表名为 `db_name.tbl_name`，或者表名匹配 `db_pattern.tbl_pattern` 模式，则丢弃表的更新。

在服务器决定是否执行前评估这些过滤规则，所以，所有事件都会发给 slave 然后进行过滤。

## 使用过滤将事件分配给 slave

那么，master 过滤和 slave 过滤各自有什么优缺点呢？乍一看，好像在 master 上使用 `binlog-*-db` 选项而不是 `replicate-*-db` 过滤事件，是构建数据库不错的想法。那样，网络就不会有太多负载，slave 上也没有很多无用事件。但是，前面讲过，master 过滤存在以下问题：

- 由于事件从二进制日志过滤，而且只有一个二进制日志，所以无法“切分”变更然后发送到各个不同服务器。
- 二进制日志还用于 PITR（即时恢复），所以如果服务器出现问题，无法进行完全恢复。
- 如果由于某些原因必须切分数据，这是不可能的，因为二进制日志已经进行了过滤，而且不可能“撤销过滤”。

理想的情况是，在 master 发送事件的时候过滤，但是写入二进制日志的时候不过滤。最好是 slave 可以控制过滤，能够决定复制哪些数据。对 MySQL 5.1 及以后的版本，这是不可能的，必须使用 `replicate-*` 选项过滤事件，即在 slave 上过滤事件。

例如，用一个专门的 slave 存储用户数据，位于 app 数据库的 users 表和 profiles 表中，关闭服务器，向 `my.cnf` 文件添加以下过滤选项：

```
[mysqld]
...
replicate-wild-do-table=app.users
replicate-wild-do-table=app.profiles
```

177

如果担心网络流量（在远程网络复制时很重要），你可以在 master 所在的机器上再配置一个 relay 服务器，如图 6-6 所示（或者配置在 master 所在的网段内），保存 master 的二进制日志过滤后的版本。

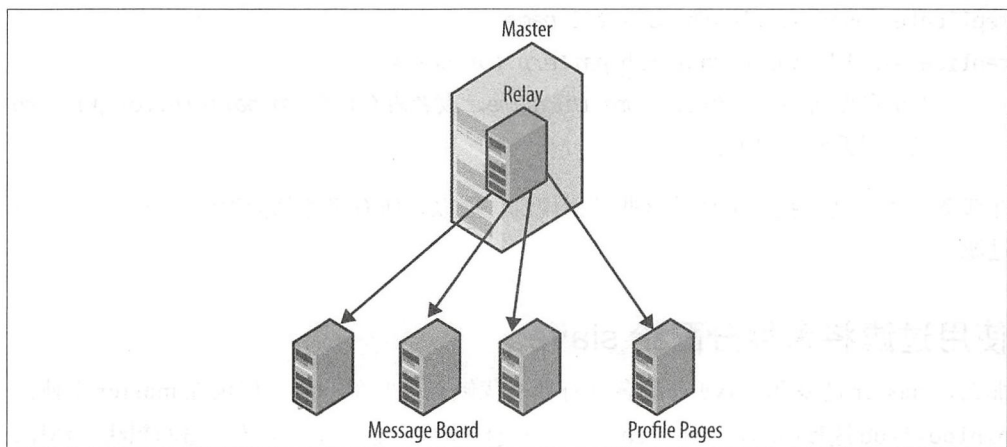


图6-6: 将master和relay放在同一台机器上进行过滤

## 数据的一致性管理

前面讨论过，异步复制的问题之一就是数据一致性管理。为了说明这个问题，假设有一个电子商务网站，用户浏览产品，并将他们想买的东西放入购物车。当用户将产品加入购物车时，这个变更向 master 发出请求。但是当 Web 服务器请求获取购物车的内容信息时，这个查询被转交给负责响应此类查询的 slave。由于 master 比 slave 超前，这个变更可能还没有到达 slave，所以转发到 slave 的查询可能看到购物车为空。这样用户当然会大吃一惊，马上将选中产品再次加入购物车，结果发现购物车中有两个产品，因为这时 slave 与 master 同步了，复制了购物车的两次变化。显然要避免这种情况发生，否则可能引起大量用户的不满。

178 为了避免数据过于陈旧，要保证 slave 提供的是有用的最新数据。如果还要添加 relay，问题就更加棘手。解决的基本思想是将每个在 master 上提交的事务做个标记，等 slave 获取到这个事务的时候（或者更晚），才在 slave 上执行查询。

随着 MySQL 5.6 引入了全局事务标识符（GTID），slave 和客户端的故障转移变得简单多了，因为大多数讨论的技术都可以自动处理了。全局事务 ID 将在第 8 章的“全局事务标识符”一节详细介绍，这里我们还是主要讲老办法，方便那些还没升级到 5.6 的用户。后面会单独比较 5.6 之前和之后版本的区别。

MySQL 5.6 之前，根据 master 和 slave 之间是否存在 relay 服务器，有不同的解决办法。

## 非级联部署的一致性

如果所有 slave 直接连接到 master, 那么一致性检查很容易。这时, 只要在事务提交后记录 binlog 位置, 然后等待 slave 达到这个位置(使用前面介绍的 MASTER\_POS\_WAIT 函数)。但是, 无法到达事务写入 binlog 的确切位置, 为什么呢? 因为在事务提交以后、SHOW MASTER STATUS 执行之前这段时间内, 可能有多事件被写入 binlog。

这并不重要, 因为没有必要达到事务写入的确切 binlog 位置, 只要能到达事务位置或其后面某个位置就够了。SHOW MASTER STATUS 命令显示当前复制正在写的事件位置, 事务提交后执行这个命令得到的 binlog 位置, 足以进行一致性检查。

示例 6-10 给出了处理更新的 PHP 代码, 保证数据不过时。

示例6-10: 避免读取过时数据的PHP代码

```
function fetch_master_pos($server) {
    $result = $server->query('SHOW MASTER STATUS');
    if ($result == NULL)
        return NULL;                                // 执行失败
    $row = $result->fetch_assoc();
    if ($row == NULL)
        return NULL;                                // 未启动 binlog
    $pos = array($row['File'], $row['Position']);
    $result->close();
    return $pos;
}
```

```
function sync_with_master($master, $slave) {
    $pos = fetch_master_pos($master);
    if ($pos == NULL)
        return FALSE;
    if (!wait_for_pos($slave, $pos[0], $pos[1]))
        return FALSE;
    return TRUE;
}
```

```
function wait_for_pos($server, $file, $pos) {
    $result = $server->query(
        "SELECT MASTER_POS_WAIT('$file', $pos)");
    if ($result == NULL)
        return FALSE;                                // 执行失败
    $row = $result->fetch_row();
    if ($row == NULL)
        return FALSE;                                // 空结果集?!
    if ($row[0] == NULL || $row[0] < 0)
```

179

```

        return FALSE;                                // 同步失败
    $result->close();
    return TRUE;
}

function commit_and_sync($master, $slave) {
    if ($master->commit()) {
        if (!sync_with_master($master, $slave))
            return NULL;                            // 同步失败
        return TRUE;                                // 提交并同步成功
    }
    return FALSE;                                    // 提交失败（未同步）
}

function start_trans($server) {
    $server->autocommit(FALSE);
}

```

示例 6-10 中有 `commit_and_sync` 函数和 `start_trans` 函数，以及三个辅助函数，即 `fetch_master_pos`、`wait_for_pos` 和 `sync_with_master`。`commit_and_sync` 函数提交事务，然后等待事务到达指定的 `slave`。它有两个输入参数，即到 `master` 的连接对象和到 `slave` 的连接对象。如果提交和同步完成，函数就返回 `TRUE`，提交失败则返回 `FALSE`，提交成功但同步失败则返回 `NULL`（可能是 `slave` 出错或者 `slave` 无法连接上 `master`）。

该函数提交当前事务以后，如果提交成功，则使用 `SHOW MASTER STATUS` 命令取得当前 `master` 的 `binlog` 位置。由于在事务提交之后、`SHOW MASTER STATUS` 命令发出之前，可能有其他线程更新了数据库，所以可能（很有可能）返回的位置不是事务结束的 `binlog` 位置，而是事务写入 `binlog` 之后的某个位置。前面提过，从一致性的角度来看，这没什么问题，  
180 因为当我们到达这个靠后的位置时，事务已经执行过了。

从 `master` 取得 `binlog` 位置以后，接着连接 `slave`，并使用 `MASTER_POS_WAIT` 函数等待 `master` 位置。如果 `slave` 正在运行，则这个函数调用会被阻塞，直到 `slave` 到达这个位置；但如果 `slave` 不在运行，立即返回 `NULL`。如果函数正在等待时 `slave` 停止运行也会发生这种情况，就像 `slave` 线程执行语句出错一样。无论哪种情况，`NULL` 表示事务尚未到达 `slave`，所以检查函数调用的结果很重要。如果 `MASTER_POS_WAIT` 函数返回 0，则说明 `slave` 已经收到这个事务了，因此同步很容易成功。

首先像往常一样连接服务器，然后要使用这些函数来启动、提交和中止事务。示例 6-11 给出了函数使用的示例，但这里不考虑错误检查，因为这依赖于错误处理的方式。



示例6-11: start\_trans 和commit\_and\_sync函数的使用

```
require_once './database.inc';
```

```
start_trans($master);
```

```
$master->query('INSERT INTO t1 SELECT 2*a FROM t1');
```

```
commit_and_sync($master, $slave);
```



PHP 脚本有最大运行时间，默认是 30 秒。如果脚本运行超过这个时间，就会终止运行。在运行示例 6-10 中的代码时，要记住要么在安全模式下运行，要么更改最大执行时间。

## 级联部署的一致性

由于 MySQL 5.6 引入了全局事务标识符，MySQL 5.6 的服务器一致性管理很简单，就像本章前面“非级联部署的一致性”那一节一样。因为事务标识符在不同机器间不改变，所以在发出事务的服务器与你想要连接的服务器之间，不管有多少个中继服务器都一样。

在 MySQL 5.6 之前，级联部署的一致性管理与简单复制拓扑结构的一致性管理大不相同，简单复制拓扑中每个 slave 都直接连接到 master。由于每个中间的中继服务器都会更改这个位置，所以无法等待一个 master 位置到达最终 slave（即级联底部的 slave）。要想其他办法等待事务到达最终 slave。为了保证读取到的数据没有过时，大体上有两种方案。 ◀ 181

第一种方案是利用全局事务标识符（参见附录 B）来提升 slave，并反复轮询 slave 有没有处理这个事务。与 MySQL 5.6 中的全局事务标识符不同，这里没有等待函数，所以需要反复轮询。

MASTER\_POS\_WAIT 函数在处理等待的时候很方便，如果能够使用这个函数，可以解决很多问题。第二种方案是用这个函数将从 master 到最终 slave 路径上的所有 relay 都连接起来，保证所有的变化都能传递到 slave，如图 6-7 所示。master 和 slave 之间的每个 relay 都要连接，因为你不知道各个 relay 要用到哪个 binlog 位置。

两种方案各有特点，下面来看看它们各自的优缺点。

如果 slave 相对于 master 来说基本是同步的，第一种方案仅对最终 slave 进行简单的检查，结果通常是事务已经被复制到 slave，然后继续处理。如果事务还没有被处理，可能在下次检查之前处理，所以在第二次检查最终 slave 时，事务就会到达 slave。如果检查的时间间隔足够短，用户就不会察觉到延迟，所以一次一致性检查在轮询最终 slave 的时候，通常需要一个或两个额外的消息。这种方法只需要轮询最终 slave，而不用轮询中间 ◀ 182

slave。从管理的角度来看，这也是一个优势，因为不需要跟踪中间 slave，也不用关心它们之间是如何连接的。

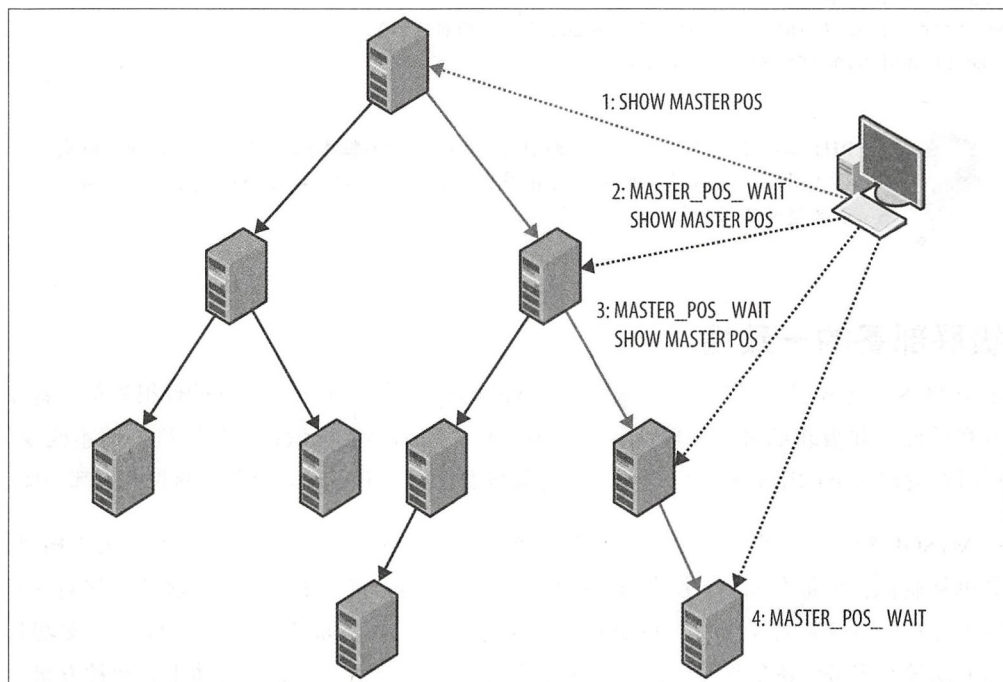


图6-7：在中继链上同步所有服务器

另一方面，如果 slave 大多是滞后的，或者各个 slave 的复制延迟差距较大，此时使用第二种方案会更好。第一个方法反复轮询 slave，然而大部分时候得到的结果是事务还没有在 slave 上提交。可以增加轮询的时间间隔，但是如果轮询间隔太大，响应时间将变得不可接受，所以第一个方案不适合。这种情况下，最好使用第二种方案，等待变更在复制树（replication tree）上向下扩散，然后执行查询。

考虑一个大小为  $N$  的树，额外请求的数目与  $\log N$  成正比。例如，如果有 50 个中继服务器，每个中继服务器处理 50 个最终 slave，那么处理 2500 个最终 slave 需要两个额外的请求：一个是发给中继服务器的请求，另一个是发给最终 slave 的请求。

第二种方案的缺点是：

- 应用程序代码需要访问 relay，这样才能轮流连接每个 relay，等待位置到达 relay。
- 应用程序代码需要知道复制的架构，才能查询 relay。

查询 relay 会拖慢速度, 因为要处理更多的事情, 但在实践中, 这或许并不是问题。通过引入一个缓存数据库连接层, 可以避免部分流量问题。每次请求来临, 缓存层会记住 binlog 位置, 只有当 binlog 位置比缓存位置大时, 才查询 relay。下面是缓存的大致代码:

```
function wait_for_pos($server, $wait_for_pos) {
    if (cached position for $server > $wait_for_pos)
        return TRUE;
    else {
        code to wait for position and update cache
    }
}
```

由于 binlog 位置总是不断增加的 (如果某个 binlog 位置已经达到, 这个位置就不会再用了), 因此不会返回不正确的结果。要确定哪个方法更有效, 唯一的方法就是监控和分析部署, 确保对应用程序来说执行查询的速度足够快。

183

示例 6-12 给出了第一种方案的代码, 即反复查询 slave, 检查事务是否执行。该代码在 master 上检查 Last\_Exec\_Trans 表 (见第 5 章中介绍), 然后在 slave 上反复读这个表, 直到发现正确的事务为止。

示例6-12: 使用轮询避免读取过时数据的PHP代码

```
function fetch_trans_id($server) {
    $result = $server->query(
        "SELECT server_id, trans_id FROM Last_Exec_Trans");
    if ($result == NULL)
        return NULL; // 执行失败
    $row = $result->fetch_assoc();
    if ($row == NULL)
        return NULL; // 空表? !
    $gid = array($row['server_id'], $row['trans_id']);
    $result->close();
    return $gid;
}
```

```
function wait_for_trans_id($server, $server_id, $trans_id) {
    if ($server_id == NULL || $trans_id == NULL)
        return TRUE; // 没有事务执行, 已同步

    $server->autocommit(TRUE); ❶
    $gid = fetch_trans_id($server); ❷
    if ($gid == NULL)
        return FALSE;
    list($current_server_id, $current_trans_id) = $gid;
    while ($current_server_id != $server_id || $current_trans_id < $trans_id) {
```

```

        usleep(500000); // 等待半秒
        $gid = fetch_trans_id($server);
        if ($gid == NULL)
            return FALSE;
        list($current_server_id, $current_trans_id) = $gid;
    }
    return TRUE;
}

function commit_and_sync($master, $slave) {
    if ($master->commit()) {
        $gid = fetch_trans_id($master);
        if ($gid == NULL)
            return NULL;
        if (!wait_for_trans_id($slave, $gid[0], $gid[1]))
            return NULL;
        return TRUE;
    }
    return FALSE;
}

function start_trans($server) {
    $server->autocommit(FALSE);
}

```

184

commit\_and\_sync 和 start\_trans 函数与示例 6-10 和示例 6-11 的原理和使用方式一样。不同的是，示例 6-12 中的函数内部调用了 fetch\_trans\_id 和 wait\_for\_trans\_id，而不是 fetch\_master\_pos 和 wait\_for\_pos。代码中有几点需要注意：

- ❶ 开始查询 slave 之前，wait\_for\_trans\_id 中打开了自动提交。如果隔离等级是 REPEATABLE READ（可重复读）或者更严格，则每次 SELECT 语句的全局事务标识符都一样，所以必须有这一步。
- ❷ 为了避免 wait\_for\_trans\_id 中不必要的 sleep，获取全局事务标识符并在进入循环之前检查一次。

这个代码只需要访问 master 和 slave，而不用访问中间的中继服务器。

示例 6-13 给出了不读取过时数据的代码，它使用查询 master 和最终 slave 之间的所有服务器的技术。首先确定最终 slave 和 master 之间的整条服务器链，然后顺着这个链依次同步每个服务器，直到事务到达最终 slave。代码重用了示例 7-8 中的 fetch\_master\_pos 和 wait\_for\_pos，这里不再重复。本代码没有实现任何缓存层的功能。



示例6-13: 通过等待避免读取过时数据的PHP代码

```
function fetch_relay_chain($master, $final) {
    $servers = array();
    $server = $final;
    while ($server !== $master) {
        $server = get_master_for($server);
        $servers[] = $server;
    }
    $servers[] = $master;
    return $servers;
}

function commit_and_sync($master, $slave) {
    if ($master->commit()) {
        $server = fetch_relay_chain($master, $slave);
        for ($i = sizeof($server) - 1; $i > 1 ; --$i) {
            if (!sync_with_master($server[$i], $server[$i-1]))
                return NULL;           // 同步失败
        }
    }
}

function start_trans($server) {
    $server->autocommit(FALSE);
}
```

使用 `fetch_relay_chain` 函数可以查找 master 和 slave 之间的所有服务器。从 slave 出发，使用 `get_master_for` 函数到达 master。这里我们故意没有给出这个函数，因为这对我们目前的讨论没有任何意义。当然这个函数还是需要定义的。

获得中继链以后，沿着这个链同步 master 和它的 slave。这是通过 `sync_with_master` 函数实现的，在示例 6-10 中定义过。

要想获取某个服务器的 master，可以使用 `SHOW SLAVE STATUS`，然后读取 `Master_Host` 和 `Master_Port` 字段。但是，如果每个要提交的事务都这么做，系统会变得很慢。

因为拓扑结构很少变更，最好将信息缓存在应用服务器上或者其他地方，这样可以避免数据库服务器的流量过大。

在第 5 章中已经学习了如何处理 master 故障，例如，将故障转移到另一个 master 上，或者将 slave 提升为 master。而且，一旦 master 修复以后，需要将其重新部署到现有的环境中。master 是部署中的关键组件，通常是比 slave 强大的机器，在重新部署时要将其恢复到

master 位置。由于 master 在没有预期的情况下停止,它很可能与部署中的其他组件不同步。有以下两种情况:

- 如果 master 离线时间较长,系统的其他部分已经提交了很多事务,而 master 不知道。从某种意义上来说,跟系统的其他部分相比, master 其实处于另一种未来 (alternative future)。图 6-8 给出了这种情况的示意图。
- 如果 master 提交了事务并将其写入二进制日志,然后在确认事务之后发送崩溃,事务可能还没有到达 slave。即 master 上有些事务没有被 slave 处理,而且系统的其他部分也不知道这些事务。

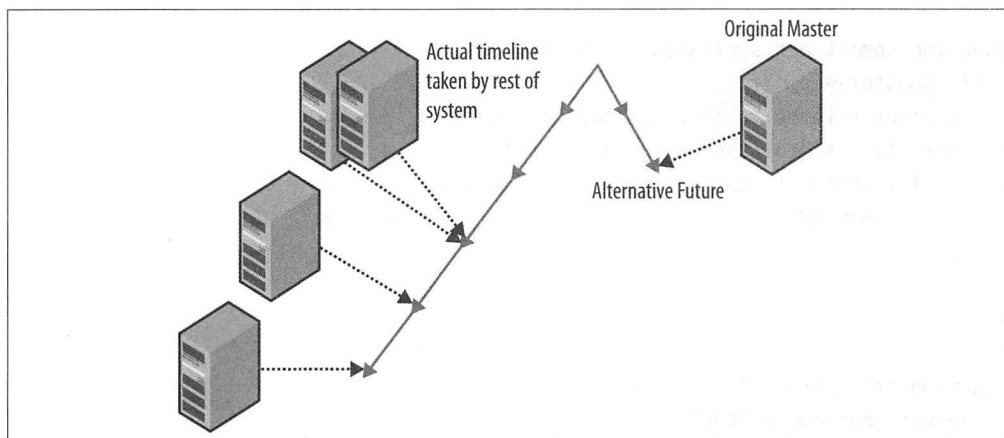


图6-8: 原来的master的另一种未来

186 如果原来的 master 没有远远落后于当前 master, 解决第一个问题最简单的方法就是将原来的 master 作为 slave 连接到当前 master, 一旦同步以后将所有 slave 切换回 master。但是, 如果原来的 master 已经离线很长时间, 更快的方法是, 克隆其中一个 slave 将其提升为 master, 然后将所有 slave 切换到 master。

如果 master 处于另一种未来, 那么新事务可能不应该带回部署中。为什么呢? 因为突如其来的新事务可能与现有的事务存在微妙的冲突。例如, 假定事务是消息板上的一条消息, 用户可能会重复提交这个消息。如果某个消息以前写过但丢了 (因为在将消息发给 slave 之前 master 发生崩溃), 突然又重新出现, 这会使用户迷惑不解, 肯定也令人讨厌。同样的道理, 由于 master 的重新部署, 导致购物车突然多了产品, 也会让用户不太高兴。

总之, 这两种不同步的问题 (即 master 处于另一种未来和 master 需要同步的情况) 都可以解决, 只需克隆一个 slave, 将其提升为 master, 然后依次将每个当前 slave 切换到这个 master 即可。



但是，这些问题说明了保证一致性是多么重要，即检查 master 上的变更在其他系统同样可用，然后才报告事务完成，以防 master 发生崩溃。本章所用的代码都假设用户立即读取数据，因此在服务器执行读查询之前检查查询是否达到 slave。从恢复的角度看，没必要这么麻烦，只需要保证事务在至少一台其他机器（比如与 master 连接的其中一个 slave 或 relay）上可用就够了。一般来说，如果变更在  $n$  个服务器上可用，可以容忍  $n-1$  次故障。

对 MySQL 5.6 来说，可以利用全局事务标识符来处理。在示例 6-10 中，直接使用 WAIT\_UNTIL\_SQL\_THREAD\_AFTER\_GTIDS 函数代替 MASTER\_POS\_WAIT 即可，这样 wait\_for\_pos 的定义就变成了：

```
function wait_for_pos($server, $gtids) {
    $result = $server->query(
        "SELECT WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS($gtids)");
    if ($result == NULL)
        return FALSE; // 执行失败
    $row = $result->fetch_row();
    if ($row == NULL)
        return FALSE; // 空结果集?!
    if ($row[0] == NULL || $row[0] < 0)
        return FALSE; // 同步失败
    $result->close();
    return TRUE;
}
```

关于全局事务标识符的更多描述参见第 8 章“全局事务标识符”一节。

## 小结

本章学习了通过横向扩展提高应用程序吞吐量的技术，即添加更多的服务器来处理更多的数据请求。我们还介绍了利用复制来建立 MySQL 横向扩展的方法，并给出了一些具体示例。在下一章中你将了解到更多的高级复制概念。

一阵敲门声使 Joel 注意到 Summerson 先生正站在他办公室门前。“我喜欢你那个横向扩展服务器的报告，Joel，我希望你马上开始着手做。可以用我们机房中多余的服务器。”

Joel 很高兴当初决定向老板提出这个建议。“好的，先生。什么时候上线？”



Summerson 先生笑了，看了一下表，说“现在还不是时候。”然后就走了。

Joel 不知道他是不是在开玩笑，所以他决定马上开始。他拿起那本 *MySQL High Availability* 的复印本和笔记，向机房走去。“希望我设置了 TiVo，”他嘀咕着，他知道这次又要熬夜了。





# 数据分片

Joel 刚刚看完服务器日志，注意到几个查询有点问题。他在工程笔记上标注需要注意这些查询，确定它们究竟只是简单的长时间查询，还是需要做些重构使它们更有效。他正在记每个查询的用户名，这时他老板发来 Jabber 消息“ping”。

“当初真不该建议使用即时通信的，”他想。Joel 回复“pong”，等老板再分配其他小任务。虽然这总好过在办公室偷袭，但 Joel 太了解老板了，他知道 Summerson 先生早晚会放弃 Jabber，继续他一贯的“路过式”日常工作。

“J，我需要你写个关于 sharding 的 WP。需要什么东西？”

Joel 花了好一会儿解码老板的话，估计老板觉得即时通信就跟发短信一样，打字比较困难，而且有些服务是按字数收费的。“好吧，看来他知道分片（sharding）了。那，什么是 WP 呢？”Joel 回复：“没问题，我马上做。您是说白皮书吗？”

过了一会儿，屏幕上出现“Ack（没错）”。

Joel 把 Jabber 窗口最小化，打开浏览器，在搜索框敲入“分片 MySQL 白皮书 pdf”，然后按下 enter 键。“只要有，我就能找到。”

上一章我们学会了如何扩展读操作，即将 slave 附加到 master 上，然后将读操作定向到 slave，写操作定向到 master。随着负载的增加，很容易添加更多的 slave，分担更多的读操作。所以很容易在读负载增加的时候实现扩展，但是写负载增加怎么办？所有写操作仍然定向到 master，如果写操作增加得太多，master 就成为系统扩展的瓶颈。这个时候你可能会问了，是不是有什么方法既能够扩展读又能够扩展写。本章我们讲述这个解决方案——分片。首先我们了解一下背景知识。

在前面几章里，数据库中的数据是完全存储在单个服务器中的，而这一章你会发现，数据库中的数据被分布在多个服务器上。为了避免混淆，我们用术语模式（schema）表



示语句中用到的名字，例如 `USE schema` 或者 `CREATE DATABASE schema`。而术语数据库 (database) 表示实际存储的所有数据的集合，与分布在多少台机器上无关。<sup>注1</sup>

例如，将数据库分解有两种方法：将某些表分别放在不同的机器上（又称为功能分割，functional partitioning），或者将某些表分割成不同的行分别存储在不同的机器上（称为水平分割，horizontal partitioning，这正是我们这一章要讨论的）。

## 什么是数据分片

大多数尝试扩展写操作的方法如图 7-1 所示，其中包括双向复制的两个 master，和一组客户端，根据需要更新的数据更改相应的 master。虽然这个架构看上去能够加倍写操作的处理能力（因为有两个 master），但实际上不是这样。写操作的花费仍然跟以前一样，因为每条语句都要执行两次：一次是从客户端接收的时候，一次是从另一个 master 接收的时候。客户端 A（或 B）的所有写操作都会被复制并执行两次，这样跟之前其实没什么两样。总之，这种双主结构并不能扩展写操作，所以需要寻找其他办法。最直接的方法是服务器之间不再有复制机制，这样它们就是完全分离的。

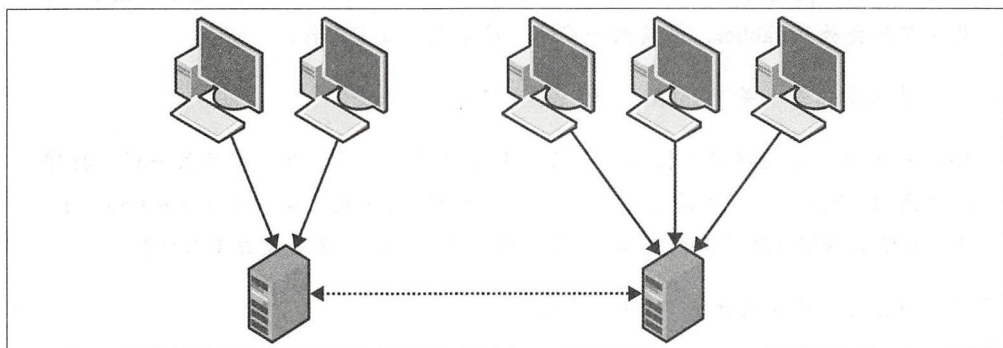


图7-1：采用双向复制的双master

通过这种结构，就可以将数据分成两个完全独立的子集，然后将客户端定向到它试图变更的数据所在的分区，从而实现扩展写操作。这样，此次更新处理并不需要其他分片消耗资源。这种分割数据的方式通常称为分片 (sharding，又称水平分割)，每个分区成为一个分片 (shard)。

注1 在 MySQL 参考手册中，模式就是数据库，这就使得概念模糊不清。实际上 SQL 标准使用“模式”这个名字，创建模式语句的语法是 `CREATE SCHEMA schema`。



## 为什么要分片

分片的原因取决于应用程序的巨大压力。分片最大的好处，也是分片最常见的原因如下：

将数据放在距离用户较近的地方

将大量数据（如图片、视频等）放在离用户近的地方可以减少延迟。这会提高系统的性能。

减少工作集（working set）的大小

如果表比较小，那么表的大部分数据甚至整张表，都可以装入内存。在内存中检索表非常高效，所以将一个表分割成很多小表或许可以极大地提高性能。也就是说，对表进行分片可以提高性能，哪怕单个服务器上有多片。

另一方面，检索表的算法在表较小的时候更有效。即使同一个机器上有多个分片，这样也能提高性能。但是，在单个机器上存储多个分片有技术限制和额外开销，所以需要在分片数目和分片大小之间权衡。

确定表的最佳大小需要监控 MySQL 服务器和 InnoDB（或其他任何存储引擎），搞清楚扫描一行平均需要多少次 I/O 操作，确定是否需要让分片更小一点。监控服务器的内容可参见第 11 章，监控 InnoDB 的内容参见第 12 章（特别是缓冲池的统计信息，这对于优化分片的大小也很重要。）

分发工作

如果数据是分片的，且这个过程够简单的话，就可以将工作并行化。如果每个分片的大小都差不多，这时并行化是最高效的。如果是为了这个目的分片数据库，就必须找到一种平衡各个分片的方法，因为分片会随着时间变大或缩小。

值得注意的是，并不是数据库中的所有数据都需要分片。可以将某些大表进行拆分，然后在每个分片上对小表做全量副本（这些通常就是全局表）。或者，也可以同时使用分片和功能分割，对庞大的数据做分片，比如文章、评论、图片和视频，而将目录和用户数据放在一个不分片的中心存储，类似图 7-2 所示的那样。



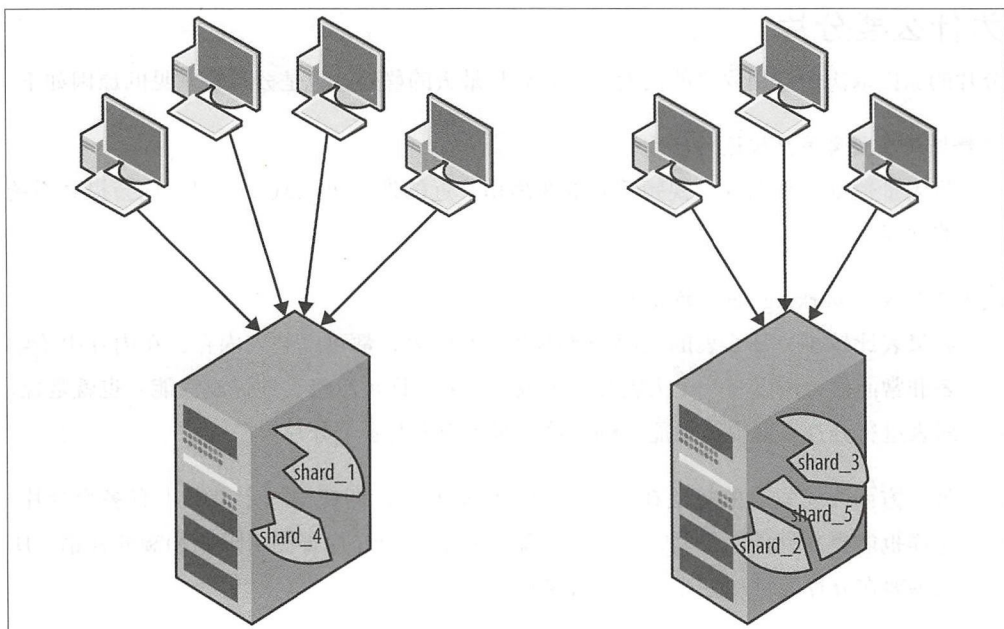


图7-2: 带有中心数据库的分片

## 分片的局限性

分片可以提高性能，但它不是万能的，也有一些局限性。本节将讨论这些局限性以及如何处理它们。

193

主要挑战在于确保在分片和不分片的数据库上进行的查询的运行结果是完全一致的，尤其在进行多表联合查询的时候就更要注意这一点了（这种情况很常见）。也就是说，需要选择一个分片索引，保证查询结果在分片或者不分片数据库上是一样的。

有些情况下，用分片索引来解决问题是不可行的甚至不可能，因为这要求重写查询（或者完全丢弃这个查询）。有两个常见问题需要解决，即跨分片连接（cross-shard joins）和 AUTO\_INCREMENT 列。下面我们分别简单介绍一下它们。

### 跨分片连接

最重要的一个局限点就是跨分片连接。由于表是分片的，所以连接属于不同分片的两张表的结果，与在一个不分片的数据库中的查询结果不可能一样。

使用跨分片连接最常见的情况是做报表。这通常需要收集整个数据库的信息，有两种方法：





- 以 map-reduce 的方式执行查询，即将查询发送到所有分片，然后将查询结果收集到单个结果集中。
- 将所有分片复制到某个单独的报表服务器，然后在报表服务器上运行查询。

以 map-reduce 的方式执行查询，好处是能够得到一个活动数据库的快照，但也意味着需要从业务应用那里占用一些资源。如果查询比较短，而且确实要求结果能够反映应用数据库的当前状态，这个方法挺适用。不过，需要监控这些查询，确保它们没有占用太多资源或者影响应用性能。

第二种方法是使用更为简单的复制技术，一般来说都采用该方法。因为大部分报表工作是在指定时间区间内完成的，是长时间任务，且与数据库的当前状态无关。

稍后，在本章后面“映射分片关键字”一节，我们介绍了一种技术，用于自动检测跨分片连接并在尝试执行这种查询的时候抛出错误。

## 使用 AUTO\_INCREMENT

我们常常会使用 AUTO\_INCREMENT 创建列的唯一标识符。但是，在分片环境中，这行不通。因为分片不会同步它们的 AUTO\_INCREMENT 标识符。也就是说，如果向一个分片插入一行，另一个分片可能也在使用相同的标识符。如果真的需要一个唯一标识符，大致有以下两种方法：

- 生成一个唯一的 UUID (<http://bit.ly/rfc-4122>)。缺点是这个标识符占用 128 个比特位（即 16 个字节）。也存在极小的可能性，同一个 UUID 出现在不同分片中，但这个可能性小到可以忽略。
- 使用复合标识符 (composite identifier)，如图 7-3 所示，前半部分是分片标识符（参见本章后面“映射分片关键字”一节），后半部分是本地生成的标识符（比如使用 AUTO\_INCREMENT 生成的）。注意生成关键字用到了分片标识符，所以如果某行的标识符被删除了，相应的分片标识符也要跟着删除。要解决这个问题，除了 AUTO\_INCREMENT 列以外，再添加一个列，用于维护分片标识符。

194

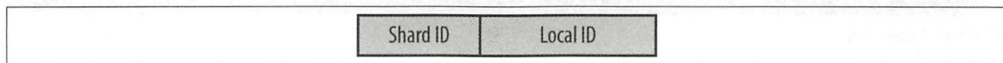


图7-3: 复合关键字



如果感兴趣的话，可以通过解决“生日问题” (<http://mathworld.wolfram.com/BirthdayProblem.html>) 的公式来计算发生冲突的概率，其中  $d$  是“天”数， $n$  是“人”数：

$$P_2(n, d) = 1 - \frac{d!}{(d-n)! d^n}$$



# 分片方案的要素

对数据库进行分片的方法最终是由用户想要执行什么样的查询决定的。例如，按年对销售数据分片是可以的（比如 2012 年的数据是一个分片，2013 是另一个分片，等等），但是如果很多用户查询都要比较各个年份的 12 月的数据，就会导致查询跨多个分片。前面讲过，跨分片连接相当麻烦，所以这会影响性能，甚至可能需要用户重写查询。

在这一小节中，我们讨论如何构建一个优秀的分片方案以及需要解决哪些问题。以下问题决定了你如何分布数据，以及如何有效地对数据进行重新分片：

- 确定如何为应用数据分区（partition）。哪些表应该分割？哪些表在所有分片上都应该有？应该在什么列上进行分片？
- 确定需要什么分片元数据（即关于分片的信息）以及如何管理元数据。包括如何将分片分配到 MySQL 服务器，如何将分片关键字映射到分片，以及“分片数据库”需要存储哪些数据。
- 确定如何分发查询。包括如何获取分片关键字将查询和事务定向到正确的分片。
- 创建分片管理的模式。包括如何监控分片负载，如何迁移分片，如何通过分割和合并分片使系统重新负载均衡。

这一章我们会逐个了解以上每一条，理解一个优秀的分片方案需要实现哪些功能。

最初，应用程序一般不会处理分片。到后来，直到数据库的增长开始影响性能的时候，就需要做一个大范围的重新设计了。所以，通常一开始是一个不分片的数据库，然后发现需要分片。我们以图 7-4 中的员工模式为例，讨论分片的要素。图中的实体表示员工的一个模式，取自 MySQL 网站上的示例模式（<http://bit.ly/ex-schema>）。为了搞清楚数据库有多大，表 7-1 列出了行数。

表7-1: 员工模式表中的行数

表	行
departments	9
dept_emp	331603
dept_manager	24
employees	300024
salaries	2844047
titles	443308

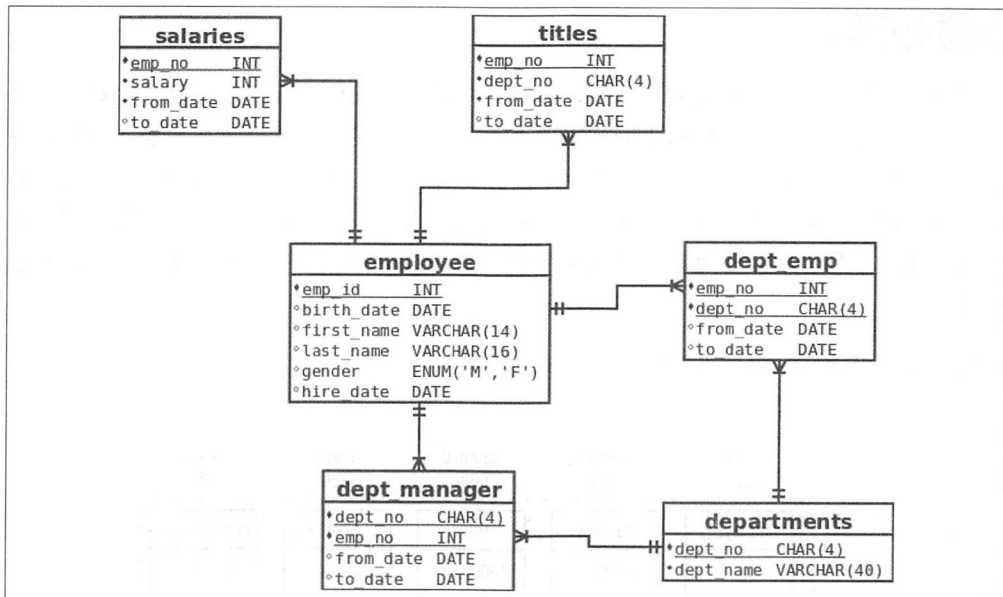


图7-4: 员工模式

## 高级分片架构

196

图 7-5 展示了分片方案的高级架构。查询来自应用程序，然后由 broker 接收。broker 决定查询将被发到哪里，可能根据一个记录分片信息的分片数据库来决定。然后，查询被发送到一个或多个应用数据库的分片上执行。broker 收集这些执行结果，有可能会对结果集进行处理，然后把这些信息发送回应用程序。

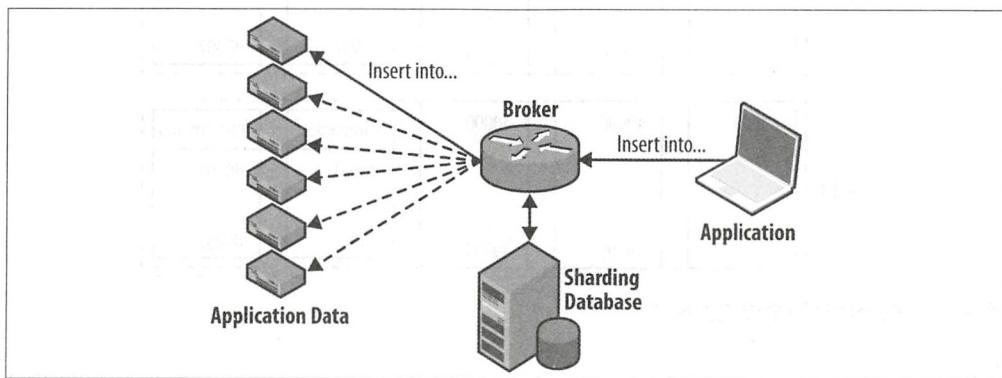


图7-5: 高级分片架构

## 数据分区

将每个数据项写入某个特定的服务器，能够有效地对写操作进行扩展。但是这对于扩展性来说还不够：高效的数据检索同样重要，为此，需要将相关数据放在一起。因此，高效分片的最大挑战是创建一个高效的分片索引（sharding index），使得经常一起访问的数据落在同一个分片上。你会发现，分片索引是定义在多个表的多个列之上的，通常每个表只用一个列，当然也可以每个表多个列。分片索引将决定哪些表需要分片，以及怎样分片。

定好分片索引之后，得到类似图 7-6 所示的结果。

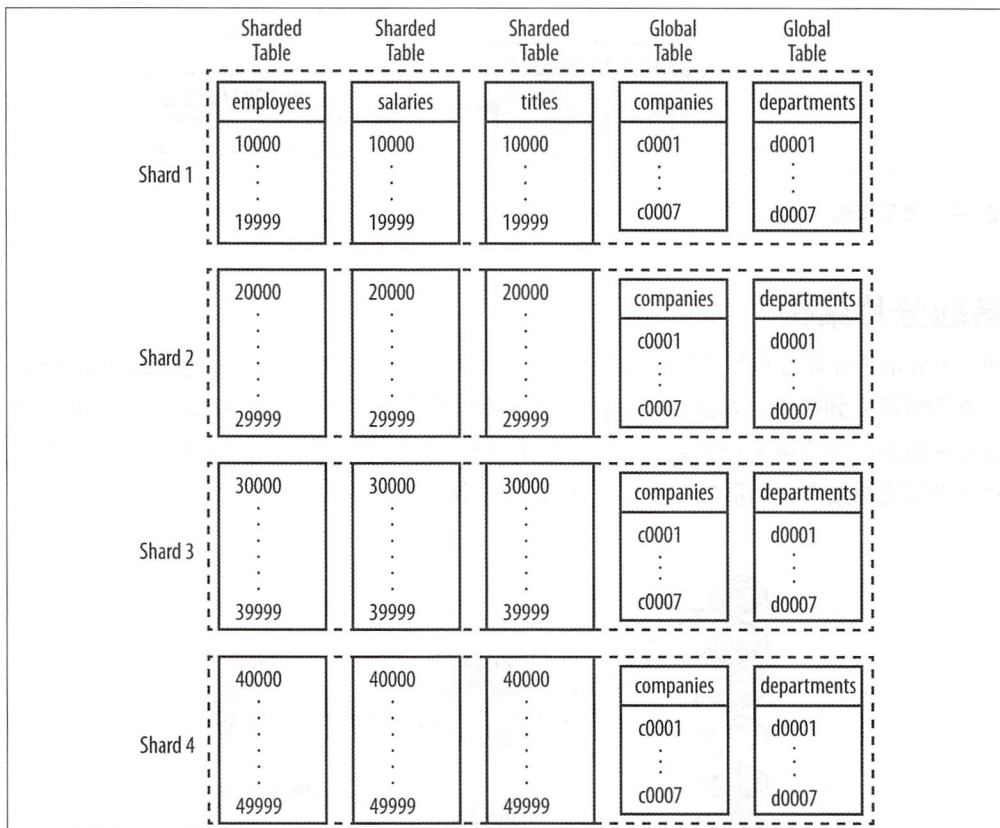


图 7-6：带有分片表和全局表的模式

我们看到，表 employees、salaries 和 titles 被分片化了，它们的行被分别存储在不同的分片上。全局表 companies 和 departments 在每个分片上都保存了一样的副本。我们将讨论如何选择列作为分片索引，以及对本例中的模式如何给出特定的解决方案。



要把数据库分割成不同的分片，需要选择一个或多个列作为分片索引，然后将行分布到不同的分片上。从单个表选择多个列作为分片索引很难维护，除非能保证正确使用。因此，通常从表中只选择一个列作为分片索引。

如果选择分片的列是主键，那就相当有利了。原因是主键列能够保证唯一索引，列中的每个值唯一确定一行。

如果选择的分片列不能保证值的唯一性，会带来问题。为了说明这一点，我们假定选择员工的国家列作为分片关键字。这时，所有属于瑞典（例如）的行存到一个分片，所有属于中国（例如）的行存到另一个分片。如果报表或更新是按照国家进行的，那么这样对数据库进行分片好像很好。但是，就算这样可行（只要每个分片都比较小的话），一旦分片变大需要进一步分割的时候，就完蛋了。这时，由于分片中的所有行都有相同的分片关键字，如果分片增长到一定程度需要再次分割，就无法再分割分片了。最后，瑞典那个分片可能有 900 万条记录，中国那个分片可能有 13 亿条记录，而且再也不能分割了。这种分配是不公平的，管理中国的服务器的负载是瑞典的 100 倍以上。

相反，如果选择表的主键（这里就是带有员工号的那列），就可以按任何方式对员工分组，创建任意大小的分区。这样，就能够得到差不多大小的分片，从而将负载均匀地分布在多个服务器上。

那么，图 7-4 所示的模式该怎样选择分片列呢？好吧，首先要回答哪些表需要分片的问题。为此，首先查看各个表有多少行，以及表之间的依赖性。

表 7-1 给出了该模式中各个表的行数。这些数字跟实际数据库分片的场景完全不一样，这个数据库完全可以放在单个服务器上，但是为了解释如何从一个不分片的数据库构建一个分片的数据库，这仍然是个不错的例子。最适合分片的表是 `employees`。不仅因为这个表很大，而且其他几个表都依赖它。稍后你会发现，如果表之间存在依赖，那么也

199

`employees` 表的主键是 `emp_no`，在这个列上分片可以将 `employees` 表的行均匀地分布在各个分片上，从而按照需求分割表。

那么，如果按照 `employees` 表的 `emp_no` 列分片，对依赖 `employees` 表的其他表会产生什么影响？`employees` 表中有外键参照，表明需要在这个列上连接其他表。看下面这个查询，获取员工职称和工资：

```
SELECT first_name, last_name, title
FROM titles JOIN employees USING (emp_no)
WHERE emp_no = employee_number
AND CURRENT_DATE BETWEEN from_date AND to_date
```

前面提过，我们的目的是保证在分片和不分片的数据库上的查询结果是一样的。由于 employees 表是按照 emp\_no 列分片的，在上面这个查询中不能关联不同分片的 titles 和 employees 列。所以，employees 表分片之后，那些不在 employees 分片中的 emp\_no 所在的 titles 行，永远不会被 employees 参照。因此，titles 也要按照 emp\_no 列分片。同理，所有 employees 有外键参照的表都要这么做，所以 employees、titles、salaries、emp\_dep 和 dept\_manager 都要进行分片。简而言之，即使只选择了一个列，也要分片多张表，每个分片列都与最初分片的 employees 表有关。

现在我们差不多对模式中所有的表都进行了分片，还剩 departments 表。这个表也要分片吗？这个表太小了，小到没有必要做分片，但如果做了分片后果如何呢？前面我们知道，这取决于从数据库获取数据的具体查询。由于 dept\_manager 和 dept\_emp 用来连接 departments 和 employees，模式在设计时就考虑了这些表的连接查询。例如，下面这个查询用来获取员工的名字和所在部门：

```
SELECT first_name, last_name, dept_name
FROM employees JOIN dept_emp USING (emp_no)
      JOIN departments USING (dept_no)
WHERE emp_no = employee number
```

这个查询比上一个 SELECT 更难分片，因为这不再是两个表共用一列的问题（即 employees 的主键），而是涉及了三个表中的两列。那么，如何确保这个查询返回的结果同不分片的查询结果一样呢？由于 employees 表是按照 emp\_no 列分片的，满足 dept\_emp.emp\_no = employees.emp\_no 和 dept\_emp.dept\_no = departments.dept\_no 条件的每一行一定在同一个分片上。如果在不同分片上，就没有匹配的行，查询将返回空结果集。

因为同一个部门的员工可能分布在不同的分片上，所以最好不要将 departments 表分片，而是把这个表作为全局表放在所有分片上。将某个表在多个分片上做副本会使表更新变得复杂（这个稍后再谈），但是由于 departments 表应该不会经常更新，所以这可能是个不错的折中。

## 自动计算可能的分片索引

本节我们提到，即使选择单个分片关键字，取决于选择对哪个表进行分片，你可能需要分片所有的表。对某一个表上的一个字段进行分片，可能会强制你对其他表的相关列也要进行分片。例如，因为 employees 表的 emp\_no 列与 salaries 表的 emp\_no 列是关联的，对 employees 表分片意味着 salaries 表也要在同一列上进行分片。

对小型模式来说，找到外键关系很简单，但是如果模式中有大量表和关系，就很难发现所有的依赖关系。使用外键仔细定义所有的依赖关系，使用MySQL的information\_schema模式，能够计算出相关列上所有可能的分片索引。找到相关列上所有集合的查询的代码如下：

```
USE information_schema;
SELECT
    GROUP_CONCAT(
        CONCAT_WS('.', table_schema, table_name, column_name)
    ) AS indexes
FROM
    key_column_usage JOIN table_constraints
    USING (table_schema, table_name, constraint_name)
WHERE
    constraint_type = 'FOREIGN KEY'
GROUP BY
    referenced_table_schema,
    referenced_table_name,
    referenced_column_name
ORDER BY
    table_schema, table_name, column_name;
```

如果在员工模式上运行这个查询，可能得到两个分片索引：

Candidate #1	Candidate #2
salaries.emp_no	dept_manager.dept_no
dept_manager.emp_no	dept_emp.dept_no
dept_emp.emp_no	
titles.emp_no	

一次查询就可以完成计算，因为每一个外键都引用了目标表的主键，这意味着参照之后不会继续有参照，否则，外键就可以参照另外一个外键了。

统计表中的行数，可以知道用哪个索引做分片更好。而且，两种方案中都有dept\_manager和dept\_emp，所以只能选择一个。

通常来说，只有一组表需要分片，比如示例7-4所示的模式。但是，在有些情况下，可能有好几“组”表需要独立进行分片。例如，假定除了员工信息，还要跟踪每个部门的出版物。员工与出版物之间是多对多的关系。在典型的数据库模式设计中，创建一个连接表，其中包含指向员工表和出版物表的外键。所以，跟踪出版物信息除了需要前面那些模式以外，还需要publications表和dept\_pub表，如图7-7所示。

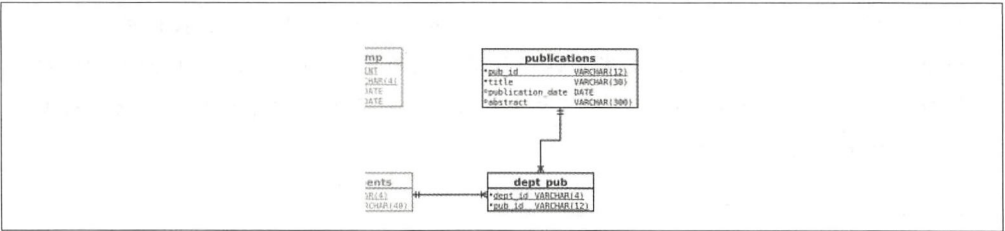


图7-7：向员工模式中添加出版物模式

如果 publications 表过大也需要分片，是可以做到的。仔细观察图 7-7 会发现，departments 表仍然在所有节点上，还有 dept\_pub 和 publications 的外键参照。也就是说，可以对 publications 和 dept\_pub 表进行分片，这样系统中就有多个独立的分片索引了。

多个独立的分片索引有什么问题呢？表 7-2 列出了不同的分片列集合，单个查询可能需要引用其中某一个集合的表加上全局表，但永远不会同时引用不同集合的表。换句话说，你可以查询员工及其职称，可以查询员工和出版物，但不能查询员工职称和出版物。

例如，这种分片方案中不能执行的查询是连接“员工”部分和“出版物”部分：

```
SELECT first_name, last_name, dept_name, COUNT(pub_id)
FROM employee JOIN dept_manager ON (emp_no)
      JOIN departments ON (dept_no)
      JOIN dept_pub ON (dept_no)
      JOIN publications ON (pub_id)
WHERE emp_no = 110386;
```

表7-2：分片索引及列

索引名称	分片列集合
si_emps	employees.emp_no、dept_emp.emp_no、salaries.emp_no、dept_manager.emp_no、titles.emp_no
si_pubs	publications.pub_no, dept_pub.pub_no

## 分配分片

想要有效地使用分片，需要以一种提升物理访问速度的方式存储它们。最直接的办法就是一个服务器上存储一个分片，当然也可以在一个服务器上保存多个虚拟分片（virtual shard）。



要知道怎样分配分片，需要回答以下问题。

应用程序使用跨模式的查询吗？

如果每个查询都总是使用单个模式（例如 employees 模式），分片就容易多了。这时可以在一个服务器上存储多个分片，每个分片保存一个模式，也不需要重写查询，因为查询总是请求单个模式。

查询根据分片方案调整吗？

如果查询是跨模式的，可以要求应用开发者在写查询的时候考虑分片方案，就仍然能够在一个服务器上存储多个分片。

这样就可以以一种可控的方式重写查询，比如可以在所有数据库名字后面加上分片号作为后缀。

需要常常重新分片吗？

如果重写查询并不容易，或是要求应用开发者以某种特定的方式写查询，就必须一个服务器保存一个分片，因为这些查询有可能是跨模式的。但是，如果常常需要重新分片（由于应用程序变更或其他原因），一个服务器一个分片可能带来性能瓶颈，所以总是需要权衡适应应用程序和保证性能。

203

如果常常需要重新分片，一个服务器上保存多个分片也是可能的，能够在服务器之间迁移分片以达到负载均衡。但是，如果某个分片变成热点，可能还是需要继续分割分片。

如何备份分片？

除了需要每次备份单个分片，还需要创建备份以在服务器间迁移分片。大多数备份方法都是对整个服务器或一个（或多个）模式创建备份。所以，要谨慎确保某个模式整体保存在同一个分片中（但是一个分片中可能有多个模式）。

## 每个服务器一个分片

最直接的办法就是在每个服务器上保存一个分片。这种情况允许跨模式查询，所以不需要重新查询。但是有两个缺点：1. 多个表可能超出服务器的主存大小，从而影响性能；2. 如果需要对这些表进行重新分片的话，那么服务器之间负载均衡工作更加昂贵。

前面提过，对数据库进行分片的目的之一是减小表的大小从而使表能够读入内存。越小的表检索时间越短，因为行数更少，而且能读入内存。

如果某个服务器过载需要减轻负载，根据前面介绍的原理我们得出的解决方案是：分割分片，要么用一个备用服务器创建一个新的分片，要么把那些不相干的行迁移到另一个

分片。如果把行迁移到某个已经存在的分片，而且每个服务器只有一个分片，迁移过来的行必须要与分片上已有的行进行合并（merge）。合并很难联机（online）完成，在一个服务器一个分片的情况下，分割和重新合并操作很昂贵。下一节，我们将讨论在迁移分片的时候如何避免合并分片。

## 每个服务器多个分片（虚拟分片）

我们已经解释过，如果能够在单个机器上保存多个分片，数据就能够更加高效地在服务器之间迁移，因为数据已经是分片的。这就为迁移分片平衡负载提供了一些灵活性，但是，如果这么做的话，就需要区分同一台服务器上的不同分片。例如，分片 123 和 234 在同一个机器上，需要区分它们各自的 `employees.dept_emp` 表。

**204** 常用的方法是在模式名上附加分片标识符。例如，分片 123 的模式 `employees` 被命名为 `employees_123`，而且每个模式分别存储每个表的一个分区，例如表 `dept_emp` 由 `employees_1.dept_emp`，`employees_2.dept_emp`，…，`employees_N.dept_emp` 组成。

由于 MySQL 服务器将每个模式存储在各自的目录下，大部分备份方法都可以对模式进行备份，但是备份单个表就会有问题。<sup>注 2</sup> 只需将不同分片的表分别存放在不同的目录，就很容易对分片备份（等会儿需要这个备份）。可以将 `replicate-do-db` 限定为服务器上的某个模式，把变更复制到单个分片上，这在服务器之间迁移分片时非常有用。

在一个服务器上保存多个分片，就更容易将一个分片迁移到另一个服务器从而减轻负载。因为一个服务器上有多个分片，甚至可以把分片迁移到其他已经拥有分片的服务器，而不需要合并这些分片的行。注意，这个方法并不能取代重新分片，毕竟还是需要一些分割分片的技术。

除了向模式名附加分片标识符以外，还可以向表名附加分片标识符。所以，名字就变成了 `employees_123.dept_emp_123`、`employees_124.dept_emp_124` 等。虽然在表上添加分片号好像有些多余，但是如果应用程序代码不小心查询了错误的分片，这时就很容易发现问题。

向模式名和（或）表名添加分片号的缺点是用户需要重写查询。如果所有查询都访问单个模式，从不执行跨模式查询，那么在将查询发送到服务器之前，通过 `USE employee_identifier` 命令很容易解决这个问题，而且旧的表名不用动。但是如果有跨模式查询，就需要重写查询，定位到所有模式名，并将分片标识符附加到每个模式名上。

向查询中插入表编号很容易出错，如果可以的话，最好泛化查询，然后自动插入正确的

注 2 也有很多备份技术能够处理单个表，但是管理和修复单个表更加复杂。用数据库构建数据库使得管理备份的工作更加容易。

表编号。例如，使用括号括住模式名中的数字，然后用一个正则表达式匹配并替换模式和表名。PHP 代码见示例 7-1。

示例7-1：替换查询中的表

```
class my_mysqli extends mysqli {
    public $shard_id;

    private function do_replace($query) {
        return preg_replace(array('/\{(\w+)\}\./', '/\{(\w+)\}/'),
            array("$1_{\$this->shard_id}.", "$1"),
            $query);
    }

    public function __construct($shard_id, $host, $user, $pass,
        $db, $port)
    {
        parent::__construct($host, $user, $pass,
            "{$db}_{\$shard_id}", $port);
        $this->shard_id = $shard_id;
    }

    public function prepare($query) {
        return parent::prepare($this->do_replace($query));
    }

    public function query($query, $resultmode = MYSQLI_STORE_RESULT) {
        return parent::query($this->do_replace($query), $resultmode);
    }
}
```

这段代码构建了 mysqli 的子类，以重写后的查询为参数重写 prepare 和 query 函数，然后调用原来的函数，传递要连接的正确数据库名。因为 mysqli 接口没有变，应用程序代码通常也就不需要改。使用这个类的例子如下所示：

```
if ($result = $mysqli->query("SELECT * FROM {test.t1}")) {
    while ($row = $result->fetch_object())
        print_r($row);
    $result->close();
}
else {
    echo "Error: " . $mysqli->error;
}
```

但是，这只有当应用开发者愿意（并且能够）向查询添加标记时才有用。而且这也很容易出错，因为应用开发者可能会忘记加标记。

206

## 映射分片关键字

上一节我们了解了当需要对表进行分片时如何选择分片列，以及如何按区间分割表。本节将深入讨论分区函数：计算正确的分片需要哪些分片元数据，以及怎样将已分片表的行映射到实际分片上。

前面解释过，映射分片关键字的目的是创建一个分区函数，以分片关键字为输入，输出某行所在的分片标识符。我们还提到，分片关键字可能不止一个，对每个分片关键字创建单独的分区函数。在这一节中，我们假定每个分片有唯一的分片标识符（即一个整数），能够用于识别上一节中的数据库和表。

在本章前面“数据分区”一节中，如果表之间存在外键关系，那么每个分区函数与多个列有关。如果想要映射某个分片关键字的值（比如“20156”），这个关键字可以来自 `employees.emp_no` 列或者 `dept_emp.emp_no` 列，因为这两个表是以相同的方式分片的。如果要把一个分片关键字的值映射到某个分片，分区函数已经暗含了列信息，而这就是关键字。

## 分片方案

分区函数可以通过静态分片方案（static sharding scheme）或者动态分片方案（dynamic sharding scheme）来实现，从名字看得出来，这些方案仅仅说明分片是可变的还是固定的。

### 静态分片方案

在静态分片方案中，通过固定不变的分配方法将分片关键字映射到分片标识符。计算分片标识符通常由连接器或者应用程序完成，而且非常高效。

例如，可以使用基于范围的分配，比如第一个分片负责第 0 到 9999 个用户，第二个分片负责第 10000 到 19999 个用户，以此类推。或者，根据标识符的最后四位数字，用这个值做哈希，从而将用户半随机地分配到分片上。

### 动态分片方案

在动态分片方案中，分片关键字通过字典查询，该字典表明哪个分片包含数据。这种方案比静态分片更加灵活，但是需要一个中心存储，称为分片数据库（sharding database）。



## 静态分片方案

你可能已经发现，如果查询分布不均匀，静态分片方案会遇到问题。例如，基于国家将行分布到不同的分片，那么“中国”分片的负载可能是“瑞典”分片的 140 倍。瑞士人当然喜欢这样，假定服务器的处理能力是一样的，他们的响应时间非常短。然而，访问中国的用户就痛苦了，因为中国分片要处理 140 倍的负载。如果哈希函数的分布不好，也会产生这种不均衡分布。所以，选择合适的分区关键字和分区函数非常重要。

示例 7-2 给出了一个静态方案的分区函数示例。

示例7-2：静态分片中字典的PHP实现

```
class Dictionary {❶
    public $shards;                                /* 我们的分片 */

    public function __construct() {
        $this->shards = array(array('127.0.0.1', 3307),
                                array('127.0.0.1', 3308),
                                array('127.0.0.1', 3309),
                                array('127.0.0.1', 3310));
    }

    public function get_connection($key, $user, $pass, $db) {❷
        $no = $key % count($this->shards);❸
        list($host, $port) = $this->shards[$no];
        $link = new my_mysqli($host, $user, $pass, $db, $port);❹
        $link->shard_id = $no;
        $link->select_db("{ $db }_{ $no }");
        return $link;
    }
}
```

\$DICT = new Dictionary('localhost', 'mats', 'xyzy', 'sharding');

- ❶ 我们定义一个 Dictionary 类，负责管理到分片系统的连接。决定使用哪个主机的全部逻辑都是在这个类里面完成的。
- ❷ 这是一个工厂方法，给定分片关键字得到一个新的连接。由于各个分片关键字可能需要查询不同的服务器，每次调用这个函数都会建立一个新的连接。
- ❸ 这个分区函数根据员工号（即分片关键字）的模计算分片。
- ❹ 使用 my\_mysqli 函数创建新的连接，这个类的定义参见示例 7-1。你也可以将其实现为从连接池获取连接。但是为了简单起见，这里不实现连接池机制。

从示例 7-2 我们知道如何使用 PHP 为静态分片创建字典。Dictionary 类用于管理到分片

系统的连接，给定分片关键字返回一个到正确分片的连接。这里我们假定分片关键字是员工号，不过该方法同样适用于任何分片关键字。在示例 7-3 中，我们给出了使用这个字典的示例，获取连接然后在分片上执行查询。

示例7-3: 使用字典的示例

```
$mysql = $DICT->get_connection($key, 'mats', 'xyzy', 'employees');
$stmt = $mysql->prepare(
    "SELECT last_name FROM {employees} WHERE emp_no = ?");
if ($stmt)
{
    $stmt->bind_param("d", $key);
    $stmt->execute();
    $stmt->bind_result($first_name, $last_name);
    while ($stmt->fetch())
        print "$first_name $last_name\n";
    $stmt->close();
}
else {
    echo "Error: " . $mysql->error;
}
```

## 动态分片方案

与静态分片方案不同，动态分片方案非常灵活。不仅允许改变分片的位置，如果必须在分片之间迁移数据，也很容易实现。不过，灵活性总是以复杂性为代价的，还可能影响性能。动态方案计算正确分片的时候需要一些额外的查询，这就增加了复杂度，也会影响性能。缓存机制可以缓存一些信息而不用每次都发送查询，这样可以减轻一些性能影响。从根本上说，要想获得好的性能，需要在考虑用户查询模式的前提下仔细地设计，这点我们后面再说。

动态分片中保存数据最简单最常见的方法是，将分片数据库以一组表的形式保存在一个分片服务器上的 MySQL 数据库中，通过查询这些表获得相关信息。示例 7-4 展示了包含每个分片信息的 `locations` 表，以及包含每个分区函数信息的 `partition_function` 表。

209 给定某个分片标识符，通过与 `locations` 表的连接查询，就能知道连接哪个服务实例。稍后讨论分片类型。

示例7-4: 动态分片中的表

```
CREATE TABLE locations (
    shard_id INT AUTO_INCREMENT,
    host VARCHAR(64),
    port INT UNSIGNED DEFAULT 3306,
    PRIMARY KEY (shard_id)
```

```
);

CREATE TABLE partition_functions (
    func_id INT AUTO_INCREMENT,
    sharding_type ENUM('RANGE','HASH','LIST'),
    PRIMARY KEY (func_id)
);
```

现在我们将示例 7-2 中 Dictionary 类的静态实现改成使用示例 7-4 中的表。在示例 7-5 中，现在 Dictionary 类从分片数据库获取分片信息，而不是静态查找。跟以前一样，使用返回的信息建立连接。你会发现，获取分片信息的查询并没有具体实现。这取决于如何设计映射，下一节再讨论这个问题。

示例 7-5：动态分片的字典实现

```
$FETCH_SHARD = <<<END_OF_QUERY
query to fetch sharding key
END_OF_QUERY;
```

```
class Dictionary {
    private $server;

    public function __construct($host, $user, $pass, $port = 3306) {
        $mysqli = new mysqli($host, $user, $pass, 'sharding', $port);
        $this->server = $mysqli;
    }

    public function get_connection($key, $user, $pass, $db, $tables) {
        global $FETCH_SHARD;
        if ($stmt = $this->server->prepare($FETCH_SHARD)){
            $stmt->bind_param('i', $key);
            $stmt->execute();
            $stmt->bind_result($no, $host, $port);
            if ($stmt->fetch()) {
                $link = new my_mysqli($no, $host, $user, $pass, $db, $port);
                $link->shard_id = $no;
                return $link;
            }
        }
        return null;
    }
}
```

## 分片映射函数

从示例 7-4 中所示的分片数据库可以看出，`partition_functions` 表中有三种分片类型。每一种分片类型使用不同的映射方法将已分片的数据映射到分片上，参见在线 MySQL 文档。这里我们讨论三种最有趣的映射。

### 列表映射

根据分片列中不重复的值的集合，将行分布到不同的分片上，比如一个国家列表。

### 区间映射

根据分片列的范围，将行分布到不同的分片上。如果分片列是 ID 列、日期，或其他很容易划分成区间的列，这种映射很方便。

### 哈希映射

根据分片关键字的值的哈希值，将行分布到不同的分片上。理论上说，这种方法能够最大程度均匀地将数据分配到分片上。

以上几种映射，列表映射最容易实现，但也最难有效地分摊负载。这种方法最适合区域性分片，确保每个分片都离它的用户比较近。区间映射容易实现，也消除了某些分摊负载的问题，但是分片之间很难达到负载均衡。哈希映射是三种方法中最能够达到负载均衡的，但也是最难有效实现的，后面我们会慢慢讲到。最重要的映射方法是区间映射和哈希映射，所以现在我们将重点讲这两种映射。

每个分片映射都要考虑两个问题：如何添加新的分片，以及如何根据分片关键字选择正确的分片。

## 区间映射

区间映射最简单的方法是，根据分片列将表中的行分到不同的区间，将每个区间分配到不同的分片。虽然区间很容易实现，问题是这些区间可能会变得零碎。而且，这个方法要求数据类型能够有效地支持区间，而并不总是这样幸运。例如，如果使用 URI 作为分片关键字，“热点”网站可能会集中在一起，而实际上我们希望它们是分散开的。这时就应该使用哈希映射获得更好的分布，具体将在本章后面的“哈希映射与一致性哈希”一节中进行介绍。

**创建索引表。**为了实现区间映射，要创建一个包含区间和映射信息的表，并将这些区间映射到分片标识符：

```
CREATE TABLE ranges (  
    shard_id INT,  
    func_id INT,
```



```

    lower_bound INT,
    UNIQUE INDEX (lower_bound),
    FOREIGN KEY (shard_id)
        REFERENCES locations(shard_id),
    FOREIGN KEY (func_id)
        REFERENCES partition_functions(func_id)
)

```

表 7-3 列出了这个表的一些典型信息，其中还包括 `partition_functions` 表的函数标识符（马上你就知道这个标识符是用来干什么的了）。每个分片只保留了下边界，因为上边界由区间中下一个分片的下边界决定。而且，分片不一定都一样大，如果在分割分片的时候同时维护上边界和下边界，是一种不必要的麻烦。表 7-3 给出了表的定义。

表7-3: 区间映射表区间

下边界	关键字	分片
0	0	1
1000	0	2
5500	0	4
7000	0	3

添加新的分片。使用基于区间的分片时，向 `ranges` 表和 `locations` 表分别插入一行，就可以添加新的分片。假如想要添加一个名为 `shard-1.example.com`、区间为 1000–2000 的分片，其区间函数由 `@func_id` 给定，首先需要向 `locations` 表中插入一行，得到一个新的分片标识符，然后使用新的分片标识符向 `ranges` 表插入一行：

```

INSERT INTO locations(host) VALUES ('shard-1.example.com');
SET @shard_id = LAST_INSERT_ID();
INSERT INTO ranges VALUES (@shard_id, @func_id, 1000);

```

注意，上边界是隐式信息，由 `ranges` 表的下一行决定。所以，添加新分片时不需要提供上边界。

获取分片。定义表并向表中填充数据以后，可以使用以下查询获取分片号、主机名，以及分片的端口。这个查询用于示例 7-5 获取分片。 ◀ 212

```

SELECT shard_id, hostname, port
  FROM ranges JOIN locations USING (shard_id)
 WHERE func_id = 0 AND ? >= ranges.lower_bound
 ORDER BY ranges.lower_bound DESC
 LIMIT 1;

```

这个查询获取所有比给定关键字的下边界低的行，并按照它们的下边界排序，然后取第一个。注意，示例 7-5 在执行查询之前先要准备查询，所以这里查询中的问号在使用的

时候将被替换成分片关键字。另一种选择是同时存储上边界和下边界，但是如果分片数或者分片区间发生变化，更新分片数据库就变得更加复杂。

## 哈希映射与一致性哈希

使用区间映射可能遇到这个问题：数据的“热点”集群无法分散，即某个分片可能过载，而且处理增加的负载需要分割分片。如果使用某个函数使数据在区间上均匀分布，负载也就被各个分片均匀分摊了。哈希函数从某个输入计算得到一个数字，这个数字称为哈希（hash）。一个优秀的哈希函数将尽可能地使输入均匀分布，输入字符串的一点小改变都会产生完全不同的输出结果。示例 7-2 中有一个最常见的哈希函数，即用来获取分片号的取模操作。

常用的哈希函数通过某种方式（例如 MD5 或 SHA-1 或更简单的函数）计算输入的哈希值，然后使用取模操作得到一个 1 到分片数之间的数据。如果需要重新分片，比方说为了添加新的分片，这个方法就不适用。这时，可能需要在分片之间迁移很多行，因为计算哈希字符串的模可能会导致所有东西都要移到新的分片。为了避免这个问题，可以使用一致性哈希（consistent hashing），保证只从一个旧分片向新分片迁移行。

为了理解这个概念，请看图 7-8。整个哈希区间（即哈希函数的输出）以环表示。分片通过某个哈希函数被分配到哈希环上的点（稍后我们讲述怎么做）。同样，行（这里用红点表示）也通过相同的哈希函数分配到环上。现在，每个分片负责该分片在哈希环上的起点到下一个分片点之间的区域。因为区域的起点可能是哈希区间的最后一个位置，然后绕到哈希区间的起点结束，所以这里用的是环状而不是直线。但是前面那些常规的哈希函数，不会发生这种情况，因为每个分片负责一段区域，而区间的终点到起点之间没有空隙。

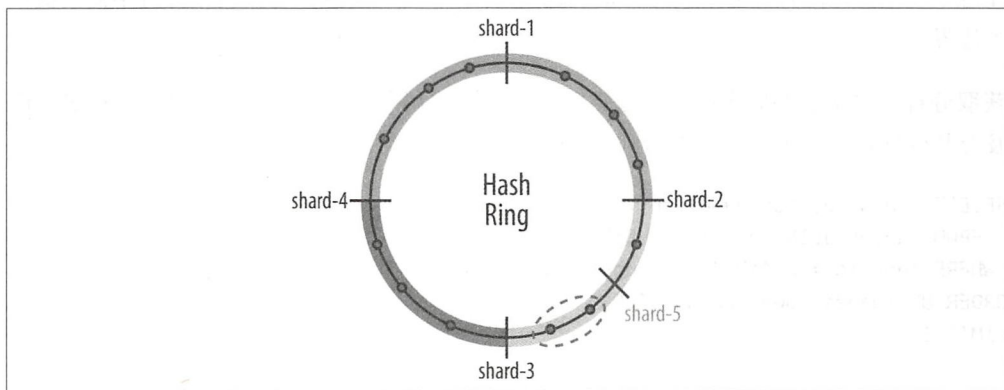


图7-8：一致性哈希中的哈希环

现在,假定要向环中添加新的分片,比如图中的 shard-5。它将被分配到环中的某个位置,比如正好要分割 shard-2,当然可能是任何一个已有分片。因为分割了 shard-2,只有旧 shard-2 上的行需要迁移到 shard-5。也就是说,新的分片只需要接管一个分片中的行并将它们迁移到新分片上,这就极大提高了性能。

那么,怎么实现一致性哈希呢?首先需要有一个优秀的哈希函数,负责生成哈希环上的值。这个哈希函数必须拥有非常大的区间,因此行能够被分配到哈希环的大量“点”上。符合需求的优秀的哈希函数来自于密码学 (Cryptography)。

密码学通过使用哈希函数为消息创造“签名”,检测是否被篡改。这些函数以任意字符串为输入,输出一个数字。密码学要求哈希函数具备大量复杂的数学属性,但对我们来说,两个最重要的属性是:提供一个包含大量比特位的哈希值,以及将输入字符串平均分布到输出区间。密码哈希函数都具备这些属性,所以是一个很好的选择。

最常用的函数是 MD5 和 SHA 家族的哈希函数 (比如, SHA-1、SHA-256/224 和 SHA-512/384)。表 7-4 列出了最常用的哈希函数以及它们输出结果的比特数。这些函数都很快,能接受任意字符串作为输入,这些函数为任意输入计算哈希值,堪称完美。

表7-4: 常用密码哈希函数

哈希函数	输出大小
MD5	128
SHA-1	160
SHA-256	256
SHA-512	512

**创建索引表。**定义一个哈希映射,需要定义一个表,存储分片所在服务器的哈希值 (通常,我们将分片的位置单独放在一个表中,所以这个表只需要存储分片标识符):

```
CREATE TABLE hashes (  
    shard_id INT,  
    func_id INT,  
    hash BINARY(32),  
    UNIQUE INDEX (hash)  
    FOREIGN KEY (shard_id)  
        REFERENCES locations(shard_id),  
    FOREIGN KEY (func_id)  
        REFERENCES partition_functions(func_id)  
)
```

添加索引能够加速检索哈希值。表 7-5 列出了这张表典型的数据内容。

表7-5: 哈希映射表hashes

关键字	分片	哈希
1	0	dfd59508d347f5e4ba41defcb973d9de
2	0	2e7d453c8d2f9d2b75a421569f758da0
3	0	468934ac4c69302a77cbe5e7fa7dc13
4	0	47a9ae8f8b8d5127fc6cc46b730f4f22

添加新的分片。要添加分片，需要向 `locations` 和 `hashes` 表中各自插入一行。构造服务器的字符串，计算得到其哈希值，就得到了 `hashes` 表要插入的行。这个代表服务器的字符串可能是一个完整的域名，当然任何表示形式都可以。例如，如果想区分不同的服务器，需要向字符串添加端口信息。哈希值存储在 `hashes` 表中，如果将函数标识符存储在 `@func_id`，用下面的语句就可以添加分片：

```
INSERT INTO locations(host) VALUES ('shard-1.example.com');
SET @shard_id = LAST_INSERT_ID();
INSERT INTO hashes VALUES (@shard_id, @func_id, MD5('shard-1.example.com'));
```

**215** 获取分片。现在表中已经有了分片信息。如果需要根据分片关键字查询分片位置，计算分片关键字的哈希值，然后找到小于这个哈希值的最大哈希值对应的分片标识符。如果没有比这个哈希值小的哈希值，则选择最大的哈希值。

```
(
  SELECT shard_id FROM hashes ❶
  WHERE MD5(sharding key) > hash
  ORDER BY hash DESC
) UNION ALL (
  SELECT shard_id FROM shard_hashes ❷
  WHERE hash = (SELECT MAX(hash) from hashes)
) LIMIT 1 ❸
```

- ❶ 这个 `SELECT` 语句选择比给定分片关键字的哈希值小的所有哈希值对应的分片。注意，这个语句可能结果为空。
- ❷ 这个 `SELECT` 语句提供一个默认值，以防前面的返回结果为空。
- ❸ 因为我们只需要一行，而 `SELECT` 语句的并集可能会得到多个分片，所以只取第一个分片。这个分片可能来自语句❶，如果语句❶返回空则来自语句❷。

## 处理查询和事务调度

现在，我们知道了如何通过选择合适的分片列对数据进行分区，如何处理分片数据（与分片配置相关的数据，比如分片位置），以及如何将分片关键字映射到分片。接下来我们需要搞清楚：



- 如何将事务分配到合适的分片
- 如何获取事务的分片关键字
- 如何使用缓存提高性能

回忆一下本章前面“高级分片架构”一节,那里提到 broker 负责将查询分配到合适的分片。这个 broker 可以是中间代理,或者由连接器实现。

将 broker 实现为中间代理,通常将所有查询发送到某个实现了 MySQL 协议的专用主机。代理从每个查询抽取分片关键字,然后将查询分配到正确的分片。使用代理作为 broker 的好处是连接器并不知道它们连接的是个代理,就好像它们在连接服务器一样。这看上去是一个透明的解决方案,但实际上并不是。对简单应用来说,这个方案很好,但是在下面“处理事务”一节你将看到,使用代理需要扩展协议,而且(或者)会限制应用程序。

◀ 216

## 处理事务

要想通过 broker 正确分配事务,需要知道待处理事务的参数。

从应用程序的角度来看,每个事务包含一个查询序列或语句序列,其中最后一个语句是提交(commit)或中止(abort)。为了理解事务是如何处理的,我们看下面这个事务,考虑该事务的每一行需要解决哪些问题:

```
START TRANSACTION; ❶
SELECT salary INTO @s FROM salaries WHERE emp_no = 20101; ❷
SET @s = 1.1 * @s; ❸
INSERT INTO salaries(emp_no, salary) VALUES (20101, @s); ❹
COMMIT; ❺
START TRANSACTION; ❻
INSERT INTO ...;
COMMIT;
```

- ❶ ❹ 在事务的开始,无法知道事务影响的数据库或表对应的分片关键字,也无法从查询中推断出来,因为根本就不会出现分片关键字。但是,START TRANSACTION 可以被延迟直到真正的语句出现,这个语句很可能包含分片关键字。但是,broker 需要知道新事务是什么时候开始的,因为可能需要切换到另一个服务器,而且对负载均衡来说知道事务的起点也很重要。
- ❷ 事务的第一个语句使这个事务看上去像一个读事务。如果这是读事务,说明该事务可以被发送到 slave 从而平衡负载。还可以从这里发现分片关键字,所以这时你就知道事务应该发送到哪个分片了。
- ❸ 设置一个用户自定义的变量,创建一个会话特定的状态。用户变量不是全局的,所以后面所有的事务都可以引用(故而依赖于)这个用户自定义变量。

- ④ 这里我们看到这是一个读写事务。如果在语句②那里认为这是一个只读事务，然后把它发给了 slave，那么现在更新的就是 slave 而不是 master。
- 如果这里发生错误，可能要中止事务表明这里出错，此时还要表明这是一个读写事务应该发送到 master，尽管事务的开头是一个 SELECT。
- ⑤ 该语句结束事务。但是如何处理会话状态呢？在这个例子中，我们设置了几个用户自定义变量，它们会保留到下一个事务吗？

从这个例子我们可以看出，代理需要处理以下几个问题：

- 为了把事务发送到正确的分片，broker 必须在看到事务的第一个语句的时候就知道分片关键字。
- 必须在将第一个语句发送到服务器之前，知道事务是只读事务还是读写事务。
- 能够推断是否处于某个事务内部，而且同一个事务的下一个语句应该使用相同的连接。
- 能够看到上一个语句是否提交了某个事务，从而确定是否切换到另一个连接。
- 确定如何处理会话特定的状态信息，比如用户自定义的变量、临时表，以及服务器变量的会话特定设置等。

理论上来说，第一个问题可以这样解决：延迟 START TRANSACTION，然后通过解析查询从该事务的第一个语句抽取分片关键字。但是，这很容易出错，而且还要求应用开发者在第一个语句中清楚地暴露分片关键字。对应用程序来说，更好的办法是在事务的第一个语句中显式提供分片关键字，要么加上特定的注释，要么允许 broker 接收频带外的分片关键字（即不作为查询的一部分）。

为了解决第二个问题，可以使用和刚才相同的方法，将事务标记为读写事务或只读事务。做法是在查询中加入特殊注释，或者将这个信息带外发送给 broker。然后，被标记为只读的事务将被发送到 slave 然后在 slave 上执行。

对第一个和第二个问题，需要检测用户什么时候出错，比如在只读事务中发出了更新语句，或者将事务发送给了错误的分片。幸运的是，MySQL 5.6 添加了 START TRANSACTION READONLY，保证应用程序不接受更新语句。检测语句是否发送给了错误的分片比较棘手。如果将示例 7-2 中的查询重写，访问错误的分片就会自动报错，因为模式名不对。如果不重写查询，就必须加入某些类似断言的功能，保证查询在正确的分片上执行。

要检测某个事务是否正在进行中，MySQL 协议的响应包中有两个标记：SERVER\_STATUS\_IN\_TRANS 和 SERVER\_STATUS\_AUTOCOMMIT。如果事务通过 START TRANSACTION 开始，第一个标记为真；但是如果 AUTOCOMMIT=0，就不设置标记。如果设置了自动提交，就设

置 `SERVER_STATUS_AUTOCOMMIT`；否则清空标记。联合使用两个标记，就能知道某个语句是否是事务的一部分，以及下一个语句是否应该使用相同的连接。目前 MySQL 连接器不支持这两个标记检查，所以只好在 broker 中跟踪事务开始的语句和自动提交标记。

如果响应包中的服务器标记表明开始了新事务，那么处理第四个问题（即检测是否有新事务开始）就变得很容易。不幸的是，目前服务器中并没有这个，所以必须监控查询，发现是否有新事务开始。记住，有些语句会隐式提交事务，通常还需要考虑这个问题。

## 分配查询

本章前面“处理事务”一节讨论过，在分片环境中处理事务是极不透明的，应用程序必须考虑是否使用分片的数据库。为此，本章介绍的简单实现目的不是为了让查询分配的过程透明化，而是为了方便应用开发者的使用。

这一节大部分内容既能够应用到代理作为 broker 的情况，也能够适用于 broker 比较接近应用程序的情况，比如在应用服务器上用 PHP 实现 broker。为了便于叙述，这一节我们采用 PHP 实现。继续使用示例 7-2 到示例 7-5，并为区间分片实现一个动态分片方案。

那么，应该要求应用开发者提供哪些信息呢？通过前面的知识我们知道，可以让应用开发者提供分片关键字。如果查询中还提供了用来分片表的分片函数的函数标识符，那么通常区间分片能够获取分片标识符，如表 7-3 所示。期望应用开发者知道查询的函数标识符是没有道理的，这样也不健壮，因为分区函数可能会改变。但是，因为查询中的每个表都是基于函数标识符进行分片的，如果查询中给定了需要访问的所有表，那么就可以推导出函数标识符。此外，还可以检查这个查询是否真的只访问基于该分区函数分区的表，可能需要参考一些全局表信息。

219

要想知道哪些表使用什么分区函数，需要引入一个新的表，保存从表到分区函数的映射。示例 7-6 给出了这个表的定义，其中每个完全限定（fully qualified）的表名对应该表使用的分区函数。如果某个表没有记录，那么它是全局表并且存储在所有分片上。

示例7-6：记录各个表及其分区函数的表

```
CREATE TABLE columns (
    schema_name VARCHAR(64),
    table_name VARCHAR(64),
    func_id INT,
    PRIMARY KEY (schema_name, table_name),
    FOREIGN KEY (func_id) REFERENCES partition_functions(func_id)
)
```

给定一组表，我们可以同时计算出分区函数标识符和分片标识符，如示例 7-7 所示。



示例7-7：从分片数据库获取分片信息的完整PHP代码

```
$FETCH_SHARD = <<<END_OF_QUERY
SELECT shard_id, host, port ❶
FROM ranges JOIN locations USING (shard_id)
WHERE ranges.func_id = (
    SELECT DISTINCT func_id
    FROM columns JOIN partition_functions USING (func_id)
    WHERE CONCAT(schema_name, '.', table_name) IN (%s) ❷
) AND %s >= ranges.lower_bound
ORDER BY ranges.lower_bound DESC LIMIT 1; ❸
END_OF_QUERY;
```

```
class Dictionary {
    private $server;

    public function __construct($host, $user, $pass, $port = 3306) {
        $mysqli = new mysqli($host, $user, $pass, 'sharding', $port); ❹
        $this->server = $mysqli;
    }

    public function get_connection($key, $user, $pass, $db, $tables) {
        global $FETCH_SHARD;
        $quoted_tables = array_map(function($table) { return "'$table'"; },
                                   $tables); ❺
        $fetch_query = sprintf($FETCH_SHARD,
                               implode(' ', $quoted_tables),
                               $this->server->escape_string($key));
        if ($res = $this->server->query($fetch_query)) {
            list($shard_id, $host, $port) = $res->fetch_row();
            $link = new my_mysqli($shard_id, $host, $user, $pass, $db, $port); ❻
            return $link;
        }
        return null;
    }
}
```

- ❶ 这个查询通过分片关键字和对应的分片表来获取分片标识符、主机，以及查询和分片对应的端口。
- ❷ 这个查询返回表的分区函数，其中每一行是一个分区函数。如果某个分区函数对应多个表，这个子查询将会返回多行。但是因为这个子查询不允许返回多行，所以会产生错误，整个查询失败，提示“子查询返回结果不止一行”。
- ❸ 这个 where 条件匹配多行（即所有下边界比给定关键字小的行）。由于只需要下边界最高的那行，按照降序对结果集合进行排序（即将下边界最高的那行放在第一位），



然后使用 LIMIT 语句只选择第一行。

- ④ 建立一个到分片数据库的连接,以便获取关于分片的信息。这里使用“普通”连接器。
- ⑤ 构建一个列表存放表,以便查找和将它们插入到语句中。
- ⑥ 通过传递必要的信息,建立一个到分片的连接。这里使用指定连接器处理查询中的模式名称替换。

## 分片管理

为了保证负载变化时系统仍然能够响应,或仅仅出于管理性目的,有时候不得不迁移数据,将整个分片迁移到不同的节点,或者在分片之间迁移数据。不论哪种情况都会带来挑战,即以最小的宕机时间重新达到负载均衡,最好是完全没有宕机。如果是自动化方案就更好了。

### 将分片迁移到其他节点

最简单的方法是将整个分片迁移到另一个节点。如果按照前面的建议,每个分片都存储在不同的模式中,那么迁移模式就像更改目录那样简单。但是,如果迁移的同时还需要处理写操作,那就完全不是一回事儿了。

在完全没有宕机时间的情况下,把分片从一个节点(源节点)迁移到另一个节点(目标节点)是不可能的,但是可以让宕机时间尽可能少。这跟第3章中创建 slave 的技术类似。主要思想是为分片做备份,在目标节点上恢复备份,然后使用复制重新执行这期间发生的任何变更。具体做法是:

1. 在源节点上创建模式的备份。在线备份或离线备份方法都可以。
2. 从前面章节我们知道,备份将数据备份到某个特定的 binlog 位置,记下这个日志位置。
3. 停止服务器,将目标节点离线。
4. 在服务器停止的过程中:
  - a. 将配置文件的 replicate-do-db 选项设置为仅复制需要迁移的那个分片:

```
[mysqld]
replicate-do-db=shard_123
```
  - b. 如果想在服务器停止的时候从源节点恢复备份,趁现在做吧。
5. 将服务器恢复运行。
6. 将复制配置为从第2步的位置开始,然后在目标服务器上启动复制。从源服务器读取事件,并将任何变更应用到将要迁移的分片上。  
保证目标节点有足够的处理能力,从而能够应对暂时增加的大量写操作。
7. 如果目标节点与源节点差距较大,锁定源节点的分片模式,阻止变更。不需要停止

目标节点上的分片变更，因为还没有写操作访问它。

最简单的方法就是发出 `LOCK TABLES` 命令，锁定分片上的所有表，当然还有其它方法。比如删除表，如果应用程序能够处理突然不见的表（稍后讲这个问题），那么这也是一个可行的方法。

8. 检查源服务器上的日志位置。因为源服务器上的分片不会再被更新，这就是恢复需要的最高日志位置。
9. 等待目标服务器同步到这个位置，使用 `START SLAVE UNTIL` 和 `MASTER_POS_WAIT`。
10. 在目标服务器上通过 `RESET SLAVE` 关闭复制。这会删除所有复制信息，包括 *master.info*、*relay-log.info* 及所有中继日志文件。如果配置复制的时候向 *my.cnf* 文件添加了配置选项，那么还要删除它们，这个最好在下一步做。
11. 这一步是可选的。将目标服务器离线，删除 *my.cnf* 文件中的 `replicate-do-db` 选项，然后再恢复服务器。  
严格来说，这一步不是必需的，因为 `replicate-do-db` 选项只是为了迁移分片而一旦迁移完成并不会影响分片的其他功能。但是，如果下一次又要迁移分片，还是要更改这个选项。
12. 更新分片信息，使得请求被定向到新的分片位置。
13. 将模式解锁，重新启动分片的写操作。
14. 删除源服务器的分片模式。取决于分片是如何锁定的，可能这个时候还存在读分片的操作，所以要考虑这个问题。

哟，步骤还真是不少啊。幸运的是，这些都可以通过 MySQL Replicant 库自动完成。根据具体应用程序的实现不同，每一步的细节都不一样。

第 15 章将讲述各种备份技术，所以这里就不列举了。注意在设计方案的时候，我们不希望处理过程与具体备份方法绑定，因为后期可能会发现其他备份方法更加适合。

为了实现上面的备份过程，需要将分片离线，即阻止任何分片的更新操作。做法是在应用程序中锁定分片，或者在模式中锁定表。

在应用程序中实现锁定需要协调所有请求，确保没有冲突。由于网页应用通常是分布式的，锁管理可能瞬间就变得非常复杂。

这里我们将问题简化为锁定单个表，即 `locations` 表，而不是将很多客户端访问的多个表都锁定。大致上来说，所有查找分片位置的操作都需要经过 `locations` 表，所以仅锁定这一个表就能够保证在迁移和重新映射分片的时候，不会开始任何分片的更新操作。但是可能有更新正在进行，比如已经在更新分片的操作，或者正要更新分片的操作。所以还要使用 `READ_ONLY` 锁定整个服务器，这样任何准备开始的更新都会被锁定并报错，

而正在进行的更新将会继续（或在某个超时后被终止）。当分片上的锁被释放后，分片就不见了，所以更新那个分片的语句就会失败，必须在新的分片上重做。

示例 7-8 自动化实现了刚刚的过程，或者也可以使用 Replicant 库。

示例7-8：在节点之间迁移分片的过程

```
_UPDATE_SHARD_MAP = ""
UPDATE locations
    SET host = %s, port = %d
    WHERE shard_id = %d
"""

def lock_shard(server, shard):
    server.use("common")
    server.sql("BEGIN")
    server.sql(("SELECT host, port, sock"
                " FROM locations"
                " WHERE shard_id = %d FOR UPDATE"), (shard,))

def unlock_shard(server):
    server.sql("COMMIT")

def move_shard(common, shard, source, target, backup_method):
    backup_pos = backup_method.backup_to()
    config = target.fetch_config()
    config.set('replicate-do-db', shard)
    target.stop().replace_config(config).start()
    replicant.change_master(target, source, backup_pos)
    replicant.slave_start(target)

    # 等待，直到 slave 至少落后 master 10 秒
    replicant.slave_status_wait_until(target,
                                      'Seconds_Behind_Master',
                                      lambda x: x < 10)

    lock_shard(common, shard)
    pos = replicant.fetch_master_pos(source)
    replicant.slave_wait_for_pos(target, pos)
    source.sql("SET GLOBAL READ_ONLY = 1")
    kill_connections(source)
    common.sql(_UPDATE_SHARD_MAP,
              (target.host, target.port, target.socket, shard))
    unlock_shard(common, shard)
    source.sql("DROP DATABASE shard_%s", (shard))
    source.sql("SET GLOBAL READ_ONLY = 1")
```

前面提过，必须记住虽然表是锁定的，但有些客户端会话可能还在使用表，因为这些客

户端已经获得了节点位置但尚未连接，或者已经在执行更新分片的操作了。

应用程序代码必须考虑这个问题。最简单的解决办法是，如果查询分片失败，应用程序重新计算节点。示例 7-9 通过更改示例 7-3 给出了如果出错重新执行查找的代码。

224

示例7-9: 更改应用程序代码处理分片迁移

```
do {
    $error = 0;
    $mysql = $DICT->get_connection($key, 'mats', 'xyzy', 'employees',
                                   array('employees.employees', 'employees.dept_emp',
                                         'employees.departments'));
    if ($stmt = $mysql->prepare($QUERY))
    {
        $stmt->bind_param("d", $key);
        if ($stmt->execute()) {
            $stmt->bind_result($first_name, $last_name, $dept_name);
            while ($stmt->fetch())
                print "$first_name $last_name $dept_name @{$mysql->shard_id}\n";
        }
        else
            $error = $stmt->errno;
        $stmt->close();
    }
    else
        $error = $mysql->errno;
    /* Handle the error */
    switch ($error) {
        case 1290: ❶
        case 1146: ❷
        case 2006: ❸
            continue;
    }
} while (0);
```

- ❶ 这时执行失败，因为服务器处于只读模式。应用程序查找分片，但在应用程序开始执行查询之前，迁移过程就开始了。
- ❷ 这时执行失败，因为模式不见了。在移动分片之前确定了分片位置，在移动分片之后尝试执行查询。

回想一下本章前面“每个服务器多个分片（虚拟分片）”一节提到的，分片标识符是模式名的一部分，因此能够发现分片不见了。如果模式名不唯一，就无法区分多个分片。



- ❸ 这时执行失败，因为连接中断。在移动分片之前确定了分片位置，并开始执行查询，但是服务器认为这个查询花费的时间太长。

## 分割分片

225

如果主机负载过大，可以将主机上的某个分片迁移到其他服务器，但是如果分片太热怎么办？答案是：分割分片。

将分片分割成更小的分片可能非常费时费力，但如果谨慎的话也可以将宕机时间减少到最小。假设我们要分割分片，并且将分片的一半内容迁移到新的节点，步骤如下：

1. 对分片中的所有模式做备份。如果使用的是在线备份方法，比如 MEB、XtraDB，或者文件系统快照等，在做备份的时候分片仍然是在线的。
2. 记下备份对应的二进制日志位置。
3. 在目标节点上恢复步骤 1 中的备份。
4. 启动从源节点到目标节点的复制。如果要避免复制过多不必要的变更，使用 `binlog-do-db` 或 `replication-do-db` 选项只复制要迁移的模式上的变更。这时，所有请求仍然访问原来的分片，而新分片是不可见的。
5. 等待复制使目标节点跟上源节点，然后锁定源分片，既不能读也不能写。
6. 等待目标主机完全与源主机同步。这时，分片的所有数据都不可用。
7. 更新分片数据库，使所有请求都被定向到新分片。
8. 解锁源分片。这时，所有数据都可用了，但是源分片和目标分片上数据有冗余。但是多出来的数据不是新分片数据的一部分，查询也不会访问这些数据。
9. 同时开启两个作业，使用常规的 `DELETE` 分别删除两个分片上多余的行。为了避免对性能带来较大的影响，可以添加 `LIMIT` 语句每次只删除几行。

## 小结

这一章介绍了通过横向扩展提高应用的吞吐量的技术，通过添加更多服务器来处理更多的数据请求。我们介绍了使用复制配置 MySQL 实现横向扩展的方法，并且提供了一些实例。下一章我们将进一步讨论高级复制概念。

Joel 感觉还不错。他当天上午向 Summerson 先生提交了他的第一份公司白皮书。他知道应该很快就有回应了。虽然老板有点高冷，但他还是指望自己的工作能立即得到检验。过了一会儿，正在他去休息室的路上，在走廊碰见了老板。“我喜欢那个关于扩展的文章，Joel。你马上就可以开始做了，楼下还有几台不用的服务器。”“嗯，马上。”Joel 笑着说，这时老板又去分配下一个路过式任务了。

226

# 深入复制

Joel 正在看邮件，一阵敲门声转移了他的注意力。原来是 Summerson 先生站在门口，Joel 一点儿也不觉得奇怪。

“什么事，先生？”

“我开始有点儿担心我们现在做的复制这玩意儿了。我希望你做点研究，看看怎样才能提高我们对它的理解。我希望你写个文档，解释一下当前的配置，以及出错的时候有哪些排除故障的具体方法，好让系统运转起来。”

Joel 就知道会有这个任务。他也开始注意到需要了解更多复制知识。“我马上就去做，先生。”

“很好，慢慢来，我希望你能够做好。”

Joel 点点头，然后老板离开了。他叹了口气，然后翻出了他最爱的 MySQL 书。他需要再仔细地看一下复制的知识。

前面几章介绍了配置和部署复制的基本知识，以确保站点的正常运行，但要想理解复制中的潜在陷阱，以及如何高效地使用复制，还需要了解复制的操作知识以及完成任务所需的各种信息。这就是本章的目标，包括以下几个方面的内容：

- 如何更加安全地将 slave 提升为 master
- 崩溃后避免数据库损坏的技巧
- 多源复制（Multisource replication）
- 基于行的复制（Row-based replication）
- 全局事务标识符（Global transaction identifiers）
- 多线程复制（Multithreaded replication）

# 复制架构基础

第4章讨论了二进制日志及审查其中记录的事件的相关工具。但我们没有讲事件是如何发给 slave 然后在那里重新执行的。一旦你理解了这些细节,就可以更大程度地控制复制,防止崩溃后带来损坏,还可以通过检查日志来审查问题。

图8-1展示了内部复制架构的示意图,其中包括几个连接到 master 的客户端、master 本身,以及若干 slave。服务器为每个连接到 master 的客户端运行一个会话 (session),该会话负责执行所有的 SQL 语句,并将结果返回给客户端。

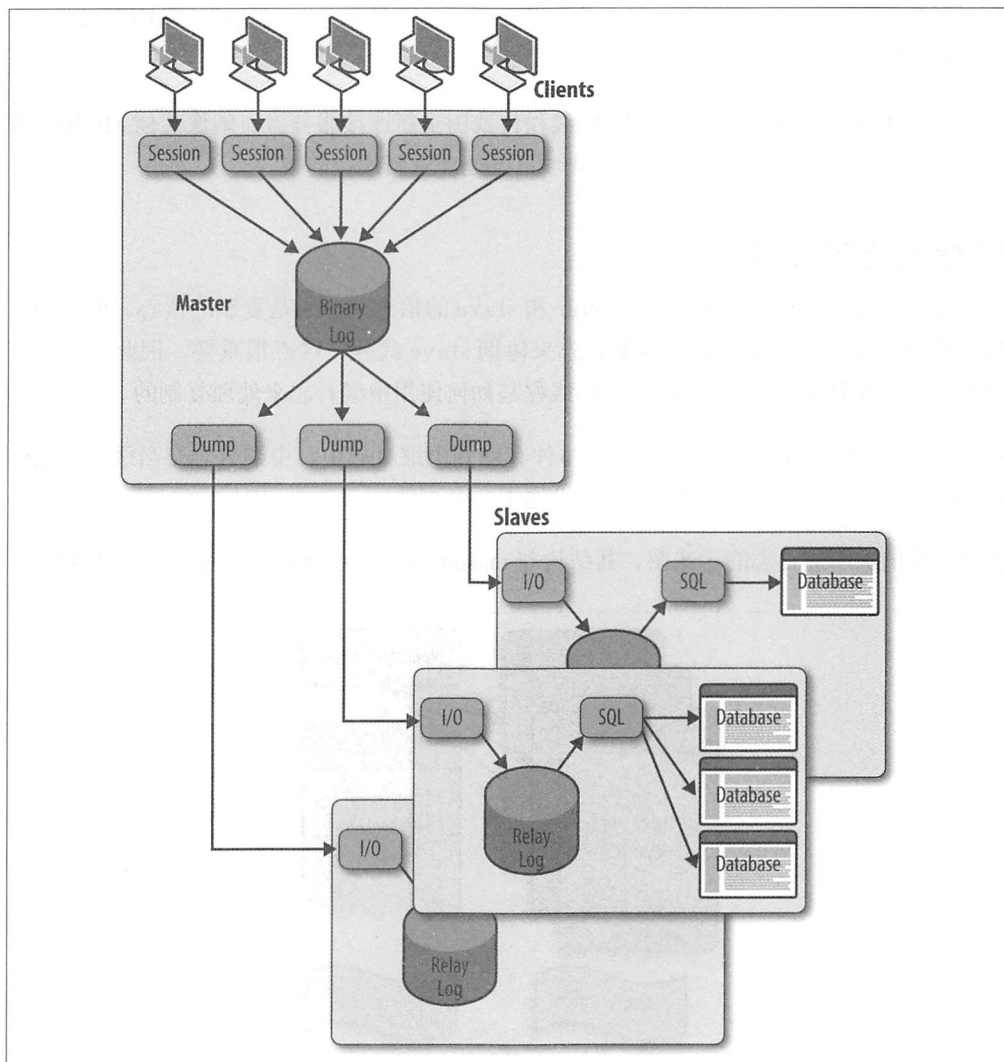


图8-1: master和若干slave的内部架构

在复制系统中，从 master 到 slave 的事件流如下所示：

1. 会话接受来自客户端的语句并执行这个语句，并与其他会话保持同步，保证每个事务的执行不与其他会话的更新发生冲突。
2. 语句执行结束之前，向二进制日志中写入一条记录，该记录包含一个或多个事件。第 3 章已经介绍了这个过程，本章不再赘述。
3. 事件写入二进制日志后，master 的转储线程（dump thread）从二进制日志中读取事件，然后将它们发送给 slave 的 I/O 线程。
4. 当 slave 的 I/O 线程接收到该事件时，将它写入中继日志（relay log）的末尾。
5. 写入中继日志后，slave 的 SQL 线程从中继日志中读取事件并执行，从而在 slave 的数据库上应用这些更新。

如果丢失了 master 连接，slave 的 I/O 线程将试图重新连接服务器，就像其他 MySQL 客户端线程一样。本章我们将看到有些选项可以处理重新连接问题。

## 229 中继日志的结构

前面已经知道，中继日志是连接 master 和 slave 的信息——它是复制的核心。明白如何使用中继日志，以及如何通过中继日志来协调 slave 线程，这点很重要。因此，这里将深入研究中继日志的结构，以及 slave 线程是如何使用中继日志来处理复制的。

230 前面讲过，I/O 线程将来自 master 的事件存储到中继日志中。中继日志充当缓冲，这样 master 不必等待 slave 执行完成就可以发送下一个事件。

图 8-2 给出了中继日志的示意图，其结构与 master 的 binlog 类似，不过多了一些文件。

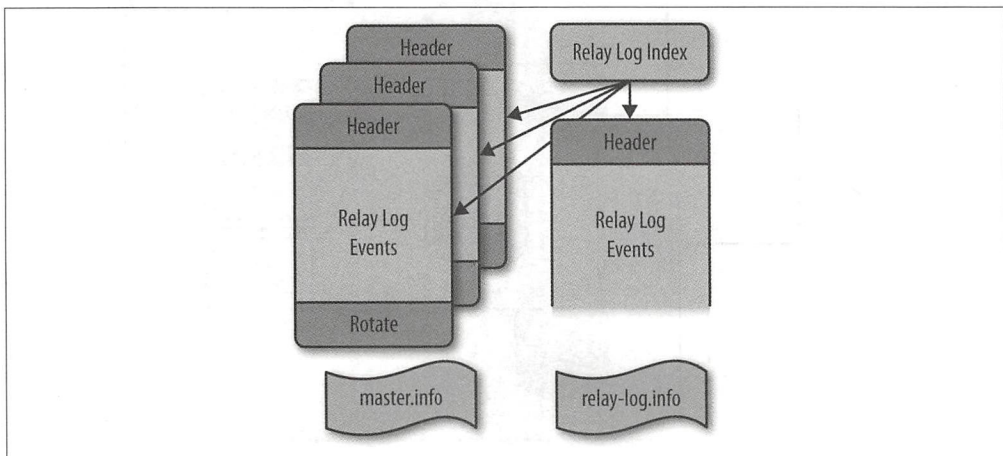


图8-2：中继日志的结构



除了二进制日志中的内容文件和索引文件以外，中继日志还维护两个文件来跟踪复制的进度，即中继日志信息文件和 master 日志信息文件。这两个文件的名字由 *my.cnf* 配置文件中的两个参数来控制：

**relay-log-info-file=filename**

这个选项设置中继日志信息文件名。也可以作为只读服务器变量 *relay\_log\_info\_file*。如果没有提供完整的文件名，则该文件位于服务器的数据目录下。默认文件名为 *relay-log.info*。

**master-info-file=filename**

这个参数设置 master 日志信息文件名。默认文件名为 *master.info*。



*master.info* 文件中的信息优先于 *my.cnf* 文件。也就是说，如果改变了 *my.cnf* 文件中的信息，然后重启服务器，将会从 *master.info* 而不是 *my.cnf* 读取信息。

231

因此，不推荐在 *my.cnf* 文件中使用 **CHANGE MASTER TO** 命令配置参数，而是直接使用 **CHANGE MASTER TO** 命令配置复制。倘若由于某种原因，需要把复制参数放到 *my.cnf* 文件，而且希望 slave 启动的时候读取这些参数，则必须在编辑 *my.cnf* 文件之前执行 **RESET SLAVE** 命令。

请谨慎执行 **RESET SLAVE**！这个命令会删除 *master.info* 和 *relay-log.info* 文件，以及所有中继日志文件！

为了便于叙述，下面的讨论中我们采用信息文件的默认文件名。

*master.info* 文件包含 master 的读位置，以及连接 master 和启动复制必需的所有信息。当 slave 的 I/O 线程启动时，如果有 *master.info* 文件，则线程从这个文件读取信息。

示例 8-1 给出了 *master.info* 文件的简单示例。我们在每行前面加了行号，而且每行末都有斜体标注（文件本身没有这些注释）。如果服务器编译不支持 SSL，则没有第 9 ~ 15 行（含有所有 SSL 参数）。示例 8-1 给出了带有 SSL 编译的参数设置。SSL 字段稍后介绍。



*master.info* 文件中记录的密码没有加密，为了保护文件，确保只有 MySQL 服务器才能读取它。通常的做法是，在服务器上定义一个专门的用户来运行服务器，将复制和数据库维护的所有文件都交给这个用户，而且只有这个用户对这些文件具备读写权限，其他用户不能访问。

示例8-1: master.info文件的内容 (MySQL 5.6.12)

```
1 23          文件中的行数
2 master-bin.000001 当前正在读取的 binlog 文件 (Master_Log_File)
3 151         最后读取的 binlog 位置 (Read_Master_Log_Pos)
4 localhost    master 连接的主机 (Master_Host)
5 root        复制用户 (Master_User)
6            复制密码
7 13000       master 所使用的端口号 (Master_Port)
8 60          slave 尝试重新连接的次数 (Connect_Retry)
9 0           如果启用了 SSL, 值为 1, 否则为 0
10           SSL 认证机构 (CA)
11           SSL CA 路径
12           SSL 证书
13           SSL 密码
14           SSL 密钥
15 0          SSL 验证服务器证书
16 60.000     心跳
17           绑定地址
18 0          忽略服务器 ID
19           Master UUID
                8c6d027e-cf38-11e2-84c7-0021cc6850ca
20 10         重试次数
21           SSL CRL
22           SSL CRL 路径
23 0          自动位置
```

如果是老版本的服务器, 格式可能略有不同。

MySQL 4.1 之前的版本没有第一行。开发人员在 4.1.1 版本中添加了行号, 这样可以为文件扩展新字段, 并通过检查行号来检测支持哪些字段。

5.1.16 版本引入了第 15 行, 即 *SSL Verify Server Certificate*, 这后面的行在 5.6 系列版本引入。

*relay-log.info* 文件跟踪复制的进度, 并由 SQL 线程负责更新。示例 8-2 给出了 *relay-log.info* 文件的一段样本, 这些行对应下一个即将执行的事件的开始。

示例8-2: relay-log.info文件的内容

```
./slave-relay-bin.000003 中继日志文件 (Relay_Log_File)
380                      中继日志位置 (Relay_Log_Pos)
master1-bin.000001      master 日志文件 (Relay_Master_Log_File)
234                      master 日志位置 (Exec_Master_Log_Pos)
```

如果某些文件不可用, 在 slave 启动的时候, 能够根据 *my.cnf* 文件中的信息及 CHANGE

MASTER TO 命令的参数，重建这些文件。



仅仅使用 *my.cnf* 文件配置 slave 并执行 CHANGE MASTER TO 命令，仍然是不够的。只有执行了 START SLAVE 命令，中继日志文件、*master.info* 文件和 *relay-log.info* 文件才会被创建。

## 复制线程

233

我们已经知道，复制需要 master 和 slave 有若干专门的线程。master 的转储线程处理 master 端的复制，而 slave 的两个线程（即 I/O 线程和 SQL 线程）处理 slave 端的复制。

### master 转储线程

当 slave I/O 线程连接 master 时，master 创建这个线程。转储线程负责从 master 的 binlog 文件读取记录，然后发送给 slave。

每个连接到 master 的 slave 都有一个转储线程。

### slave I/O 线程

这个线程负责连接 master 并请求所有 master 上的更新转储到中继日志中，以便 SQL 线程进行进一步处理。

每个 slave 有一个 I/O 线程。一旦连接建立，这个线程就一直都在，这样 slave 就能立即收到 master 的所有更新。

### slave SQL 线程

这个线程读取中继日志中的更新，然后在 slave 数据库上应用这些更新。这个线程负责协调其他 MySQL 线程，保证这些更新不与 MySQL 服务器上的其他活动产生冲突。

从 master 的角度来看，I/O 线程不过是一个能够在 master 上执行转储和 SQL 语句的客户端线程。也就是说，客户端可以连接到服务器，像 slave 一样请求 master 转储二进制日志中的更新。这就是 *mysqlbinlog* 程序（第 4 章中已经详细介绍过）的工作原理。

SQL 线程好比数据库的一个会话。也就是说，它像会话一样维护状态信息，但仍有一些区别。因为 SQL 线程需要处理 master 上不同线程的更新（master 上的所有线程的事件都会按照提交的顺序写入二进制日志），所以 SQL 线程还保存了一些额外信息以正确区分这些事件。例如，由于临时表是会话特定的，所以要将不同会话的临时表分开，将会话 ID 添加到事件中去。然后 SQL 线程根据会话 ID 保证 master 的不同会话执行各自的任务。

关于 SQL 线程如何执行事件的细节将在本章后面进行介绍。

234



I/O 线程比 SQL 线程快很多，因为 I/O 线程仅仅向日志写事件，而 SQL 线程必须知道如何在数据库上执行变更。因此，在复制的过程中，通常有些事件会缓存在中继日志中。如果 master 发生崩溃，在连接新的 master 之前，还要先处理这些事件。

为了避免丢失这些缓存事件，要等待 SQL 线程处理完毕，slave 才能尝试重新连接新的 master。

后面将介绍几种方法来检查中继日志是否为空，或者是否还有事件未执行。

## 启动和停止 slave 线程

第 3 章已经介绍了如何使用 `START SLAVE` 命令启动 slave，但忽略了很多细节。现在我们将深入描述 slave 线程的启动和停止。

服务器启动时，如果存在 `master.info` 文件，还会同时启动 slave 线程。前面提到，如果服务器做了复制配置，并且使用 `START SLAVE` 命令启动了 slave 上的 I/O 线程和 SQL 线程，就会创建 `master.info`。所以，如果上一个会话是复制会话，则从 `master.info` 和 `relay-log.info` 文件中存储的最后位置开始恢复复制，其中两个 slave 线程略有不同。

### slave I/O 线程

slave I/O 线程从 `master.info` 文件读取最后读位置进行恢复。

在写事件时，I/O 线程将轮换 (rotate) 中继日志文件，即当到达指定文件容量后将开始写入一个新文件，并更新相应的位置信息。

### slave SQL 线程

slave SQL 线程从 `relay-log.info` 文件读取中继日志位置进行恢复。

使用 `START SLAVE` 命令启动 slave 线程，`STOP SLAVE` 命令停止 slave 线程。这些命令可以控制 slave 线程，还可以用来停止和启动 I/O 线程和 SQL 线程：

### `START SLAVE` 和 `STOP SLAVE`

启动或停止 I/O 线程和 slave 线程。

### `START SLAVE IO_THREAD` 和 `STOP SLAVE IO_THREAD`

仅启动或停止 I/O 线程。

235

### `START SLAVE SQL_THREAD` 和 `STOP SLAVE SQL_THREAD`

仅启动或停止 SQL 线程。



停止 slave 线程时，复制的当前状态将保存到 *master.info* 和 *relay-log.info* 文件中。然后 slave 线程再次启动时读取这些信息。



如果使用 `master-host` 参数（可以在 *my.cnf* 文件中设置，或者启动 *mysqld* 时加上这个参数）指定 master 主机，也会启动 slave。

但是不推荐直接使用 `master-host` 参数，而是在 `CHANGE MASTER` 命令中附加 `master_host` 参数，所以这里就不介绍这个参数了。

## 通过 Internet 运行复制

在分隔两地的数据中心之间进行复制的原因有很多。其中一个原因是灾难恢复，例如地震或停电等。还可以将用户定向到较近的站点，例如内容分发网络（content delivery networks），从而提高响应速度。尽管资源充足的组织租得起专用光纤，我们还是假定使用开放的 Internet 来连接。

master 发给 slave 的事件总是不安全的：事实上，很容易解码这些事件，并从中获得复制的信息。如果在防火墙内，并且不通过 Internet 复制（例如在两个数据中心之间进行复制），可能就够安全了。可是一旦你需要跨镇或洲的数据中心进行复制，这时通过加密来保护信息不被窃取就变得尤为重要。

Internet 上的数据传输一般使用 SSL 进行加密。保护数据有以下几种方法，它们基本上都需要用到 SSL：

- 使用服务器内置的加密支持，对 master 到 slave 的复制进行加密。
- 对于不支持 SSL 的程序，使用 Stunnel 程序建立一个 SSL 隧道（其实就是一个虚拟的私有网络）。
- 在隧道模式（tunnel mode）下使用 SSH。

最后一种看上去并不比使用 Stunnel 有优势，但如果机器上不能安装新程序且服务器无法启用 SSH，这种方法就非常有用。这时可以使用 SSH 建立一个隧道，我们不深入讨论这种方法。

使用内置的 SSL 支持或 Stunnel 建立安全连接时，需要：

236

- 权威认证机构（CA）的证书
- 服务器的（公有）证书
- 服务器的（私有）密钥

关于如何生成、管理和使用 SSL 证书的具体内容超出了本书的范畴，不过为了说明问题，示例 8-3 简单演示了如何生成自签名的公有证书及其私有密钥。这个例子假定使用 OpenSSL，其配置文件位于 `/etc/ssl/openssl.cnf`。

示例8-3：生成自签名的公有证书及其私有密钥

```
$ sudo openssl req -new -x509 -days 365 -nodes \
    -config /etc/ssl/openssl.cnf \
> -out /etc/ssl/certs/master.pem -keyout /etc/ssl/private/master.key
Generating a 1024 bit RSA private key
.....++++++
.+++++
writing new private key to '/etc/ssl/private/master.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:SE
State or Province Name (full name) [Some-State]:Uppland
Locality Name (eg, city) []:Storvreta
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Big Inc.
Organizational Unit Name (eg, section) []:Database Management
Common Name (eg, YOUR name) []:master-1.example.com
Email Address []:mats@example.com
```

这个证书签名过程将自签名的公有证书放在 `/etc/ssl/certs/master.pem`，私有密钥放在 `/etc/ssl/private/master.key`（也用于为公有证书签名）。

同样，slave 也需要创建服务器密钥和服务器证书。为了便于讨论，我们将 `/etc/ssl/certs/slave.pem` 作为 slave 服务器的公有证书名，`/etc/ssl/private/slave.key` 作为 slave 服务器的私有密钥名。

## 237 使用内置支持建立安全复制

加密 master 和 slave 之间的连接，最简单的方法就是使用支持 SSL 的服务器。关于如何编译支持 SSL 的服务器超出了本书的范畴，如果感兴趣，请查阅在线参考手册。

要使用内置的 SSL 支持，做法是：

- 配置 master，使 master 的密钥可用。

- 配置 slave，加密复制通道。

将 master 配置为支持 SSL，需要向 *my.cnf* 文件添加以下参数：

```
[mysqld]
ssl-capath=/etc/ssl/certs
ssl-cert=/etc/ssl/certs/master.pem
ssl-key=/etc/ssl/private/master.key
```

ssl-capath 提供了可信 CA 证书所在的目录文件名，ssl-cert 给出了服务器证书的文件名，ssl-key 提供了服务器私有密钥的文件名。同样，更新 *my.cnf* 文件以后必须重启服务器。

现在 master 已经配置好了，可以为任何客户端提供 SSL 支持。由于 slave 使用的是普通客户端协议，所以还要配置 slave 也能使用 SSL。

为了配置 slave 使用 SSL 连接，执行带有 MASTER\_SSL 参数的 CHANGE MASTER TO 命令开启 SSL 连接，然后执行 MASTER\_SSL\_CAPATH、MASTER\_SSL\_CERT 和 MASTER\_SSL\_KEY 命令，这些命令的功能与刚刚提到的配置参数 ssl-capath、ssl-cert 和 ssl-key 类似，不同的是这些命令用来设置连接的 slave 端：

```
slave> CHANGE MASTER TO
-> MASTER_HOST = 'master-1',
-> MASTER_USER = 'repl_user',
-> MASTER_PASSWORD = 'xyzyz',
-> MASTER_SSL_CAPATH = '/etc/ssl/certs',
-> MASTER_SSL_CERT = '/etc/ssl/certs/slave.pem',
-> MASTER_SSL_KEY = '/etc/ssl/private/slave.key';
Query OK, 0 rows affected (0.00 sec)
```

```
slave> START SLAVE;
Query OK, 0 rows affected (0.15 sec)
```

现在已经建立了 slave 到 master 之间的安全通道。

## 使用 Stunnel 建立安全复制

◀ 238

Stunnel 是一个简单易用的 SSL 隧道应用，可以配置为 SSL 服务器或 SSL 客户端。

使用 Stunnel 建立安全连接与使用内置支持建立 SSL 连接差不多，只是需要更多的配置。如果服务器不支持 SSL 编译，或者由于某种原因你想把加密和解密数据所需的额外处理从 MySQL 服务器分离出去（只有在多核 CPU 的情况下才有意义），这种方法非常有用。

和内置支持一样，该方法也需要 CA 证书，而且每个服务器都要有公有证书和私有密钥。

然后，stunnel 命令需要这些信息，而服务器不用。

图 8-3 中有一个 master、一个 slave 和两个通过不安全网络通信的 Stunnel 实例。slave 服务器端的 Stunnel 实例通过标准的 MySQL 客户端连接，接收来自 slave 服务器的数据，加密后发送给 master 服务器端的 Stunnel 实例。master 服务器端的 Stunnel 实例监听 SSL 端口，接收加密数据后将其解密，然后通过客户端连接将解密后的数据发送给 master 服务器的非 SSL 端口。

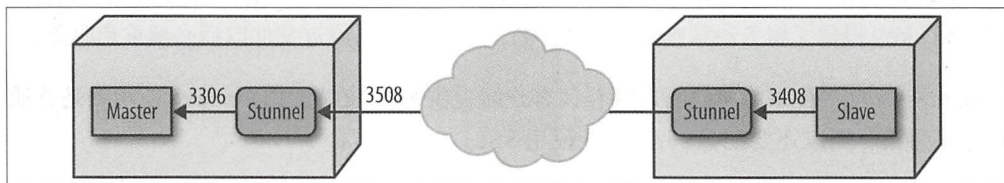


图8-3：使用Stunnel在不安全信道上复制

建立 Stunnel 所需的配置文件如示例 8-4 所示，其中 SSL 连接的监听套接字（socket）为 3508，master 服务器监听的 MySQL 套接字默认为 3306。证书和密钥文件均使用前面例子中的文件名。

示例8-4：master服务器配置文件/etc/stunnel/master.conf

```
cert=/etc/ssl/certs/master.pem
key=/etc/ssl/private/master.key
CApath=/etc/ssl/certs
[mysqlrepl]
accept = 3508
connect = 3306
```

客户端建立 Stunnel 所需的配置文件如示例 8-5 所示，其中 3408 端口为中间端口（即 slave 连接 Stunnel 的本地非 SSL 端口），然后 Stunnel 连接 master 服务器的 3508 端口，如上面示例 8-4 所示。

示例8-5：slave服务器配置文件/etc/stunnel/slave.conf

```
cert=/etc/ssl/certs/slave.pem
key=/etc/ssl/private/slave.key
CApath=/etc/ssl/certs
[mysqlrepl]
accept = 3408
connect = master-1:3508
```

现在每台服务器都可以启动 Stunnel 程序了，然后配置 slave 连接 slave 服务器端的



Stunnel 实例。由于这个 Stunnel 实例运行在 slave 所在的服务器上，所以将 localhost 设置为要连接的 master 主机，其连接的端口号为 3408。然后由 Stunnel 管理与 master 服务器的连接隧道。

```
slave> CHANGE MASTER TO
-> MASTER_HOST = 'localhost',
-> MASTER_PORT = 3408,
-> MASTER_USER = 'repl_user',
-> MASTER_PASSWORD = 'xyzzzy';
Query OK, 0 rows affected (0.00 sec)
```

```
slave> START SLAVE;
Query OK, 0 rows affected (0.15 sec)
```

这样就完成了在不安全网络中建立安全连接。



如果操作系统是基于 Debian 的 Linux（如 Debian 或 Ubuntu），向 *etc/default/stunnel4* 文件添加 **ENABLED=1**，可以为每个配置文件启动一个 Stunnel 实例（配置文件位于 */etc/stunnel* 目录）。

所以，如果按照本节介绍的方法创建 Stunnel 配置文件，无论什么时候启动机器，都会自动启动一个 slave Stunnel 实例和一个 master Stunnel 实例。

## 细粒度控制复制

理解了复制的内部原理和复制所用的信息，就可以更好地控制复制，知道如何避免可能遇到的问题。本节将介绍一些有用的背景知识。

### 关于复制状态的信息

关于复制状态的信息大部分都在 slave 上，但 master 上也有一些。master 上的信息大多与 binlog（见第 4 章）相关，当然还有连接的 slave 的相关信息。

240

SHOW SLAVE HOSTS 命令仅显示使用 report-host 参数的 slave 信息，slave 使用 report-host 参数告诉 master 服务器的连接信息。master 不能相信这些连接 slave 的信息，因为 master 和 slave 之间还有 NAT 路由器。除了主机名以外，以下参数也提供了关于连接 slave 的信息：

report-host

连接中的 slave 主机名，通常是 slave 的域名或其他类似的标识符，实际上可以是任意字符串。示例 8-6 中我们使用的名字是“Magic Slave”。

report-port

slave 用于监听连接的端口，默认是 3306。

report-user

连接 master 的用户。该值不需要与 CHANGE MASTER TO 的值一致。只有服务器使用了 show-slave-auth-info 参数时才有这个参数。

report-password

连接 master 的密码。该值不需要与 CHANGE MASTER TO 的密码一致。

show-slave-auth-info

如果启用了这个参数，SHOW SLAVE HOSTS 命令将同时输出用户和密码的相关信息。

示例 8-6 给出了 SHOW SLAVE HOSTS 命令的输出示例，这里 master 连接了三个 slave。

示例8-6: SHOW SLAVE HOSTS命令的输出示例

master> SHOW SLAVE HOSTS;

Server_id	Host	Port	Rpl_recovery_rank	Master_id
2	slave-1	3306	0	1
3	slave-2	3306	0	1
4	Magic Slave	3306	0	1

1 row in set (0.00 sec)

241 这个命令列出了连接 master 的 slave 及其相关信息，包括通过中继间接连接 master 的 slave。如果启用了 show-slave-auth-info 参数，还会多输出两个字段（这里我们没有列出）。

这些字段仅用来提供信息，不一定表示真实的 slave 主机或端口，用户和密码也不一定就是 CHANGE MASTER TO 配置 slave 时所需的真实用户和密码。

Server\_id

已连接的 slave 的服务器 ID。

Host

report-host 参数指定的主机名。

User

report-user 参数指定的用户名。

Password

report-password 参数指定的密码。

Port

端口号。

Master\_id

指定 slave 从哪个服务器进行复制。

Rpl\_recovery\_rank

该字段从不使用, MySQL 5.5 版本已经去掉了这个字段。



间接连接的 slave 的信息并不完全可信, 因为添加 slave 的时候信息可能不准确。

所以, 需要去掉间接连接的 slave 信息, 只显示直接连接的 slave, 这样信息就可信了。

可使用 SHOW MASTER LOGS 命令查看 master 的二进制日志文件。该命令的典型输出参见示例 8-7。

示例8-7: SHOW MASTER LOGS命令的典型输出

master> SHOW MASTER LOGS;

```
+-----+-----+
| Log_name          | File_size |
+-----+-----+
| master-bin.000011 | 469768    |
| master-bin.000012 | 1254768   |
| master-bin.000013 | 474768    |
| master-bin.000014 | 4768      |
+-----+-----+
1 row in set (0.00 sec)
```

SHOW MASTER STATUS 命令 (参见示例 8-8) 给出下一个事件即将写入二进制日志的位置。由于 master 只有一个 binlog 文件, 所以该表总是只有一行记录。因此, SHOW MASTER LOGS 输出的最后一行正好与该命令的输出匹配, 只是显示的字段不同。也就是说, 如果需要使用 SHOW MASTER LOGS 命令来实现某些功能, 不用执行 SHOW MASTER STATUS, 只要参照 SHOW MASTER LOGS 的最后一行即可。

242

示例8-8: SHOW MASTER STATUS命令的典型输出

```
master> SHOW MASTER STATUS;
```

```
+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+-----+-----+-----+-----+
| master-bin.000014 |      4768 |              |                  |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

SHOW SLAVE STATUS 命令用于查看 slave 线程的状态。该命令几乎包含了复制状态的所有信息。下面仔细地看看这个命令的输出, 如示例 8-9 所示。

示例8-9: SHOW SLAVE STATUS命令的典型输出

```
Slave_IO_State: Waiting for master to send event
Master_Host: master1.example.com
Master_User: repl_user
Master_Port: 3306
Connect_Retry: 1
Master_Log_File: master-bin.000001
Read_Master_Log_Pos: 192
Relay_Log_File: slave-relay-bin.000006
Relay_Log_Pos: 252
Relay_Master_Log_File: master-bin.000001
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
Replicate_Do_DB:
Replicate_Ignore_DB:
Replicate_Do_Table:
Replicate_Ignore_Table:
Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
Last_Errno: 0
Last_Error:
Skip_Counter: 0
Exec_Master_Log_Pos: 192
Relay_Log_Space: 553
Until_Condition: None
Until_Log_File:
Until_Log_Pos: 0
Master_SSL_Allowed: No
Master_SSL_CA_File:
Master_SSL_CA_Path:
Master_SSL_Cert:
Master_SSL_Cipher:
Master_SSL_Key:
```



```
Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
      Last_IO_Errno: 0
      Last_IO_Error:
Last_SQL_Errno: 0
Last_SQL_Error:
```

## I/O 线程和 SQL 线程的状态

Slave\_IO\_Running 和 Slave\_SQL\_Running 两个字段分别表示 slave 的 I/O 线程或 SQL 线程是否正在运行。如果 slave 线程不在运行，可能是线程已经停止，或者复制过程出错。

如果 I/O 线程不在运行，Last\_IO\_Errno 和 Last\_IO\_Error 字段将给出线程停止的原因。同样，Last\_SQL\_Errno 和 Last\_SQL\_Error 字段将给出 SQL 线程停止的原因。如果线程在没有出错的情况下停止——比如，显式停止线程，或者达到了 until 条件——这时没有出错信息，error 字段为 0，类似示例 8-9 那样。其中 Last\_Errno 和 Last\_Error 字段是 Last\_SQL\_Errno 和 Last\_SQL\_Error 的同义词。

Slave\_IO\_State 描述了当前正在运行的 I/O 线程的状态。随着 I/O 线程的状态不同，其消息变化的状态如图 8-4 所示。

这些消息的含义如下：

244

等待 master 更新

在 I/O 线程初始化之后、试图建立 master 连接之前，将短暂出现这个消息。

连接 master

245

在 slave 正在尝试连接 master、但连接尚未建立时，出现这个消息。

检查 master 的版本

在 slave 已经连接上 master、正在与 master 进行握手（handshake）时，出现这个消息。

在 master 上注册 slave

在 slave 试图在 master 上注册时出现这个消息。注册时，slave 将 report-host 的值发送给 master。通常这个值是 slave 的主机名或 IP 地址，但也可以是任意字符串。master 不能仅仅依靠检查 TCP 连接的 IP 地址，因为 master 和 slave 之间可能有 NAT（网络地址转换）路由器。

请求 binlog 转储

当 slave 开始向 master 请求 binlog 转储时出现这个消息，slave 将 binlog 文件、binlog 位置和服务 ID 发送给 master。

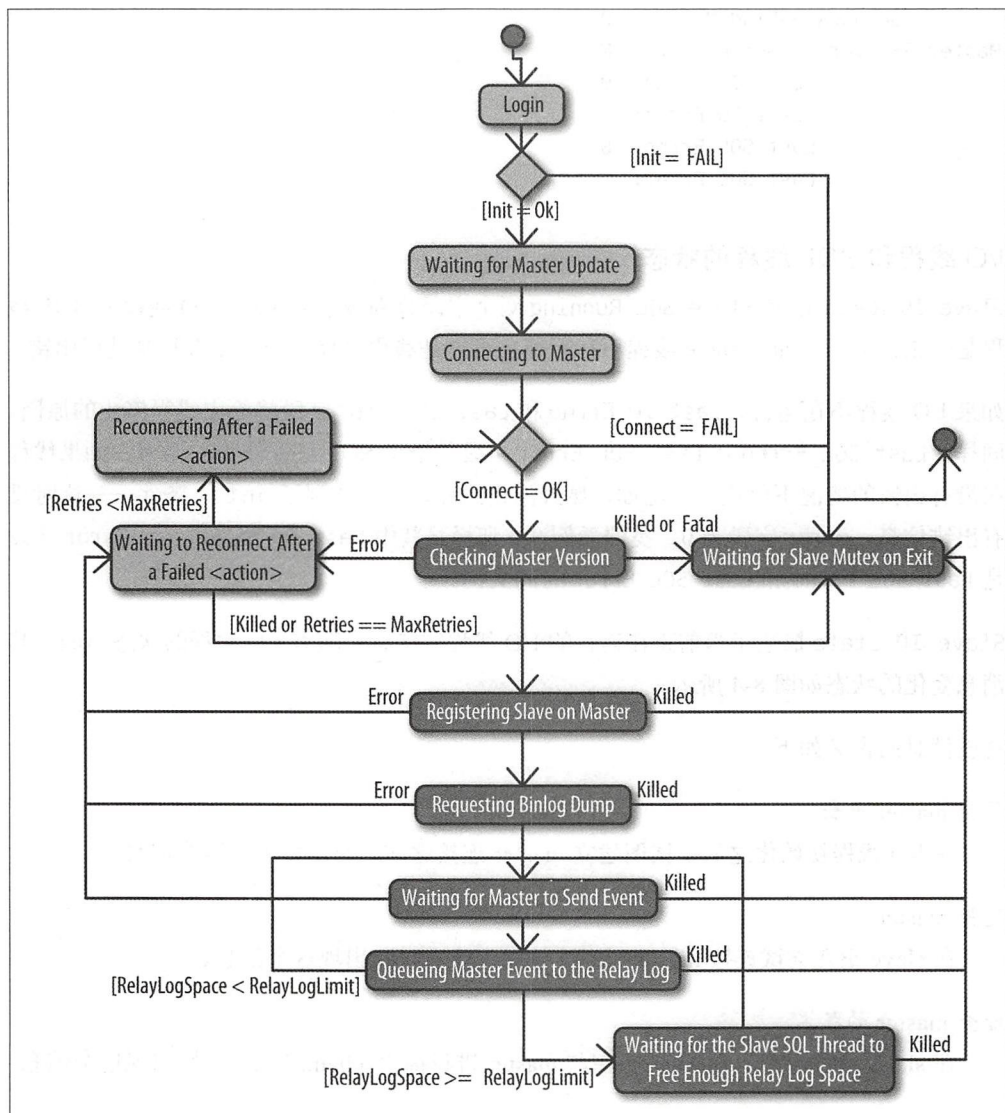


图8-4: slave I/O线程的状态

等待 master 发送事件

当 slave 已经连接上 master、并等待 master 发送事件时出现这个消息。

master 事件排队等待写入中继日志

当 slave I/O 线程正要 will master 发送的事件写入中继日志时出现这个消息。不管事件是否真的被写入中继日志还是被跳过，都会出现这个消息，其规则参见第 6 章“过滤复制事件”一节。



注意这个消息的拼写，是“Queueing”而不是“Queuing”。

使用脚本或其他工具查看这些消息时，弄清楚消息的真实含义非常重要，而不要仅仅看字面意思。

#### action后等待重新连接

当上一次 action 出错失败、slave 尝试重新连接时出现这个消息。这些操作的值可能是：

registration on master

尝试在 master 上注册

binlog dump request

从 master 请求 binlog 转储

master event read

等待或读取 master 的事件

246

#### action失败后重新连接

当 slave 操作失败后尝试重新连接 master、但尚未建立连接时出现这个消息。这些操作的值与“action后等待重新连接”消息的值一样。

#### 等待 slave 互斥体退出

关闭 I/O 线程时出现该消息。

#### 等待 slave 的 SQL 线程释放中继日志空间

当中继日志的空间大小达到限定值（由 relay-log-space-limit 参数设置）时出现这个消息，SQL 线程需要处理中继日志以便写入新事件。

## 二进制日志位置和中继日志位置

slave 上的复制在处理事件时，需要同时维护三个位置。

使用 SHOW SLAVE STATUS 命令查看这些位置，如示例 8-9 所示，其中包含以下几对字段。

#### Master\_Log\_File 和 Read\_Master\_Log\_Pos

master 的读位置：I/O 线程即将从 master 二进制日志读取的下一个事件的位置。

这些字段值来自 *master.info* 文件的第 2 行和第 3 行，如示例 8-1 所示。

#### Relay\_Master\_Log\_File 和 Exec\_Master\_Log\_Pos

master 的执行位置：SQL 线程即将执行的 master 二进制日志的下一个事件的位置。

这些字段值来自 *relay-log.info* 文件的第 3 行和第 4 行，如示例 8-2 所示。

#### Relay\_Log\_File 和 Relay\_Log\_Pos

中继日志的执行位置：SQL 线程即将执行的 slave 中继日志中下一个事件的位置。

这些字段值来自 *relay-log.info* 文件的第 1 行和第 2 行，如示例 8-2 所示。

通过这些位置可以获得复制过程的相关信息，也可以用来优化第 5 章中的某些算法。

247 例如，通过比较 master 的读位置和 master 的执行位置，可以确定是否有事件在等待执行。如果 I/O 线程已经停止，只要等待中继日志变空就行了：一旦这两个位置相同，中继日志就没有等待的事件，这时就可以安全地停止 slave 并将其重定向到新的 master。

示例 8-10 给出了等待 slave 中继日志变空的示例代码。MySQL 提供了一个方便的函数 MASTER\_POS\_WAIT，负责等待 slave 中继日志处理完所有等待中的事件。如果 slave 线程不在运行，则 MASTER\_POS\_WAIT 返回 NULL，捕获并产生异常。

示例 8-10：等待中继日志变为空的 Python 脚本

```
from mysql.replicant.errors import Error
```

```
class SlaveNotRunning(Error):
    pass

def slave_wait_for_empty_relay_log(server):
    result = server.sql("SHOW SLAVE STATUS")
    log_file = result["Master_Log_File"]
    log_pos = result["Read_Master_Log_Pos"]
    running = server.sql(
        "SELECT MASTER_POS_WAIT(%s,%s)", (log_file, log_pos))
    if running is None:
        raise SlaveNotRunning
```

使用这些位置还可以优化第 5 章中描述的场景。示例 8-21 将 slave 提升为 master 之后，在将 slave 切换到新 master 前，可能还要处理其他 slave 中继日志的事件。而且，确保被提升的 slave 在允许其他 slave 连接之前已经执行了所有事件，这样可以将数据丢失减到最小。

修改示例 5-5 中的 `order_slaves_on_position` 函数得到示例 8-11，保证切换前的 slave 已经执行了所有中继日志中的事件。该代码使用示例 8-10 中的 `slave_wait_for_empty_relay_log` 函数等待中继日志变空，然后读取 slave 位置。



示例8-11：提升slave时将事件丢失减少到最小

```
from mysql.replicant.commands import (
    fetch_slave_position,
    slave_wait_for_empty_relay_log,
)

def order_slaves_on_position(slaves):
    entries = []
    for slave in slaves:
        slave_wait_for_empty_relay_log(slave)
        pos = fetch_slave_position(slave)
        gtid = fetch_gtid_executed(slave)
        entries.append((pos, gtid, slave))
    entries.sort(key=lambda x: x[0])
    return [ entry[1:2] for entry in entries ]
```

248

除了这里所介绍的技术外，有些书中还提到了另一种技术，即使用 SHOW PROCESSLIST 命令检查 SQL 线程的状态。如果 State 字段为“已经读取所有中继日志，等待 slave I/O 线程更新”，那么 SQL 线程已经读取了全部中继日志。这个 State 消息只能由 SQL 线程产生，所以你可以放心地在所有线程中查找这个消息。

## 处理断开连接的选项

由图 8-4 可知，I/O 线程负责维护 slave 与 master 之间的连接，其中包含很多复杂的逻辑。

如果 I/O 线程丢失了 master 连接，则进行有限次尝试重新连接 master。无响应时间、重试的时间间隔和重试次数由以下三个选项控制：

**--slave-net-timeout**

可接受的无响应时间，超过这个时间之后 slave 认为 master 连接已经丢失并尝试重新连接。如果已经检测到连接已经断开，那这个参数就没有用了。这时，slave I/O 线程立即尝试重新连接（等待的时间取决于 master-connect-retry 的值，且尝试次数不得超过 master-retry-count）。

默认值为 3600 秒。

**--master-connect-retry**

两次尝试间隔的秒数。可以将这个参数设为 CHANGE MASTER TO 命令的 CONNECT\_RETRY 参数值。*my.cnf* 文件不再使用这个参数。

默认值为 60 秒。

--master-retry-count

最大尝试次数。

默认值为 86400。

这些默认值可能不是最好的，所以最好提供自定义的值。

## 249 > slave 如何处理事件

复制的核心是日志事件：它们是复制系统的信息携带者，包含所有必需的元数据，保证复制能够在 master 上执行变更从而得到一个 master 的副本。因为 master 的二进制日志是按照 master 上的事务提交顺序记录的，这些事务在 slave 上也会以同样的顺序执行，得到与 master 相同的结果。

slave SQL 线程顺序执行来自 master 的所有会话的各个事件。这种 slave 执行事件的方式带来的后果是：

slave 响应是单线程的，而 master 是多线程的

日志事件在 slave 上以单线程执行，但在 master 上是多线程的。所以，如果在 master 上提交了很多事务，slave 就难以与 master 保持同步。

有些语句是会话特定的

master 上的有些语句是特定于会话的，在 slave 上单线程执行的时候可能产生不同的结果：

- 每个用户变量都是特定于会话的。
- 临时表是特定于会话的。
- 有些函数也是特定于会话的，例如 CONNECTION\_ID。

二进制日志决定了执行顺序

尽管二进制日志的事务看上去相互独立（理论上可以并行执行），但实际上并不是这样。也就是说，slave 必须按顺序执行事务，才能保证 master 和 slave 一致。

## 管理 I/O 线程

尽管大部分的事件处理由 SQL 线程完成，但是在 SQL 线程看到这些事件之前，需要 I/O 线程做一些管理工作。所以在讨论 SQL 线程“真正执行”之前，先来看看 I/O 线程的处理。为了获得较高的处理速度，I/O 线程只使用某些字节来判断事件的类型，然后对中继日志采取必要的行动：

### 停止事件

该事件表明 slave 链中的下一个服务器被有序地停止。I/O 线程忽略这个事件，不把这个事件写入中继日志。

### 轮换 (Rotate) 事件

如果 master 上的二进制日志被轮换，中继日志也要被轮换。中继日志轮换的次数可能比 master 多，但是每次 master 的二进制日志轮换时，中继日志都要轮换。

### 格式化描述事件

中继日志轮换时保存这种事件。回想一下，两个连续的 binlog 文件的格式可能不同，所以 I/O 线程需要保存该事件以正确地处理文件。

如果是环形复制，或者双主配置（即只在两个服务器之间进行环形复制），事件将一直在环中传送下去直到到达最初发送它们的那个服务器。为了避免事件在环中无休止地复制，必须将已经执行过的事件删除。

为此，每个服务器需要检查事件是否包含该服务器本身的 ID。如果有，说明这个事件原来就是这个服务器发送的，已经在整个环上复制了一圈。为了避免事件无限复制（因此也会无限执行），不把这个事件写入中继日志，直接忽略它。设置 `replicate-same-server-id` 参数可以关闭这种检查。如果设置了这个参数，服务器将不再检查服务器 ID，这样不管服务器 ID 是什么，事件都会写入中继日志。

## SQL 线程的处理

slave SQL 线程读取中继日志，然后在 slave 上重新执行 master 的数据库语句。有些事件还需要 SQL 语句以外的特殊信息，包括：

### 将 master 的上下文发送给 slave 服务器

有时候 slave 需要状态信息才能正确执行语句。第 4 章中提到过，master 通过写一个或多个上下文事件来传递这些额外信息。有些信息是属于线程特定的，但不同于第二种信息。

### 处理不同线程的事件

由于 master 执行的事务来自多个会话，slave SQL 线程必须知道事件是由哪个线程产生的。master 了解每个语句，它会标记那些线程特定的事件。例如，通常 master 将操作临时表的事件标记为线程特定的。

### 过滤事件和表

SQL 线程负责 slave 上的过滤处理。MySQL 提供了数据库过滤（通过 `replicate-`

do-db 和 replicate-ignore-db 设置) 和表过滤 (通过 replicate-do-table、replicate-ignore-table、replicate-wild-do-table 和 replicate-wild-ignore-table 设置)。

### 跳过事件

要恢复复制, 重启复制时可以选择跳过事件。SQL 线程处理事件的跳过。

## 上下文事件

master 上的有些事件需要上下文才能正确执行。上下文通常是指线程特定的信息, 比如用户自定义的变量, 也可以是正确执行所需的状态信息, 比如含有 autoincrement 字段的表的自动增量值。要将上下文发送给 slave, master 需要向二进制日志写入一组上下文事件。

master 在写包含实际变更的事件之前, 先写上下文事件。目前, 只有 Query 事件有上下文事件, 而且上下文事件在 Query 事件之前写入二进制日志。

上下文事件可以分为以下几类:

### 用户变量事件

该事件包含用户自定义变量的变量名及变量值。

不论语句中有没有引用用户自定义变量, 都会产生这个事件。

```
SET @foo = 'SmoothNoodleMaps';  
INSERT INTO my_albums(artist, album) VALUES ('Devo', @foo);
```

### 整型变量事件

该事件存储 INSERT\_ID 或 LAST\_INSERT\_ID 会话变量的整型值。

INSERT\_ID 整型变量事件用于向含有 AUTO\_INCREMENT 列的表插入语句, 它传递 AUTO\_INCREMENT 列的下一个值。例如, 下面这个表的定义和插入语句需要该信息:

```
CREATE TABLE Artist (id INT AUTO_INCREMENT PRIMARY KEY, artist TEXT);  
INSERT INTO Artist VALUES (DEFAULT, 'The The');
```

当语句使用 LAST\_INSERT\_ID() 函数时, 就会产生 LAST\_INSERT\_ID 整型变量事件, 例如:

```
INSERT INTO Album VALUES (LAST_INSERT_ID(), 'Mind Bomb');
```



## Rand 事件

如果语句中调用了 `RAND()` 函数，则这个事件中含有随机种子，允许 slave 重新产生在 master 上生成的“随机”值。

```
INSERT INTO my_table VALUES (RAND());
```

这些上下文事件能够处理上面描述的情况，但有些情况无法通过上下文事件解决。例如，复制系统不能处理用户自定义的函数（user-defined function, UDF），除非 UDF 是确定性的，并且 slave 上也有这个函数，这时可以用用户变量事件解决。

用户变量事件的用途还有：避免非确定性函数的复制问题、提高性能，以及完整性检查等。

例如，假定要将文档存入数据表。使用 `AUTO_INCREMENT` 为每个文档自动分配一个编号。为了维护文档的完整性，还添加了文档的 MD5 校验和。表定义语句如示例 8-12 所示。

示例8-12：带有MD5校验和的文档表定义

```
CREATE TABLE document(
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  doc BLOB,
  checksum CHAR(32)
);
```

将文档及其校验和存储在一张表中，能够验证文档的完整性，保证文档不被损坏，如示例 8-13 所示。尽管目前 MD5 校验和并不是绝对安全的，但能够避免某些随机错误，比如硬盘和内存问题等。

示例8-13：向表中插入记录，并检查文档的完整性

```
master> INSERT INTO document(doc) VALUES (document);
Query OK, 1 row affected (0.02 sec)
```

```
master> UPDATE document SET checksum = MD5(doc) WHERE id = LAST_INSERT_ID();
Query OK, 1 row affected (0.04 sec)
```

```
master> SELECT id,
  ->      IF(MD5(doc) = checksum, 'OK', 'CORRUPT!') AS Status
  ->      FROM document;
```

```
+-----+-----+
| id | Status |
+-----+-----+
|  1 | OK     |
|  2 | OK     |
|  3 | OK     |
|  4 | OK     |
```

```

| 5 | OK |
| 6 | OK |
| 7 | CORRUPT! |
| 8 | OK |
| 9 | OK |
| 10 | OK |
| 11 | OK |
+-----+
11 row in set (5.75 sec)

```

但是怎么复制呢？这取决于你怎么使用校验和。示例 8-13 中执行 `INSERT` 语句时，该语句会被写入二进制日志，也就是说，slave 会重新计算 MD5 校验和。那么，如果文档在发给 slave 的时候就损坏了怎么办？那样的话，就会对已损坏的文档重新计算 MD5 校验和，就不会发现文档已经损坏。所以示例 8-13 中的语句对于复制来说是不安全的。当然，有更好的办法。

按照示例 8-14 那样改写示例 8-13，将校验和存在一个用户自定义的变量中，然后在 `INSERT` 语句中使用这个变量。由于用户自定义的变量存的是 MD5 函数计算而得的实际值，所以即使文档在传输过程中损坏（当然校验和并没有在传输时损坏），master 和 slave 也是完全一样的。不管怎样，总会发现复制时的文档是否损坏。

示例 8-14：向表中插入文档的复制安全（Replication-safe）方法

```

master> INSERT INTO document(doc) VALUES (document);
Query OK, 1 row affected (0.02 sec)

master> SELECT MD5(doc) INTO @checksum FROM document WHERE id = LAST_INSERT_ID();
Query OK, 0 rows affected (0.00 sec)

master> UPDATE document SET checksum = @checksum WHERE id = LAST_INSERT_ID();
Query OK, 1 row affected (0.04 sec)

```

## 线程特定的事件

前面提过，有些语句是特定于线程的，在不同线程中执行时可能产生不同的结果。原因是：

读写线程本地对象

不同线程的线程本地对象（thread-local object）的名字可以完全一样。典型的例子是临时表或用户自定义变量。

254

我们已经知道复制如何处理用户自定义变量，所以这一小节仅讨论临时表的处理。

使用具有线程特定结果的变量或函数

有些变量和函数由于在不同的线程中运行导致值不同。典型的例子是服务器变量 `connect_id`。

服务器对这两种情况的处理有些相同。此外，有时候复制并不区分服务器和客户端，所以结果可能稍有不同。

处理线程本地对象需要线程本地存储 (thread-local store, TLS)，由于 slave 是单线程执行，所以 TLS 是单独管理的。为了处理临时表，slave 基于服务器进程 ID、线程 ID 和线程的序列号，为临时表创建唯一的文件名。即，示例 8-15 中的两个语句（由 master 上的不同客户端运行）在 slave 上创建两个不同的文件代表临时表。

示例 8-15：两个线程分别创建一个临时表

```
master-1> CREATE TEMPORARY TABLE cache (a INT, b INT);
Query OK, 0 rows affected (0.01 sec)
```

```
master-2> CREATE TEMPORARY TABLE cache (a INT, b INT);
Query OK, 0 rows affected (0.01 sec)
```

由于 master 上的所有线程的所有语句都是按顺序存储在二进制日志中的，所以需要区分这两个语句。否则，在 slave 上执行时会出错。

为了区分二进制日志中的语句，防止它们发生冲突，服务器将包含语句的 Query 事件标记为线程特定的，并将线程 ID 添加到该事件。实际上，所有 Query 事件都有线程 ID，但只有线程特定的语句才真的有必要这么做。

当 slave 接收到一个线程特定的事件时，设置一个特定的复制 slave 线程变量，即 *pseudothread ID*，对应事件的线程 ID。然后使用这个 pseudothread ID 创建临时表。创建文件名的时候会用到 slave 服务器的进程 ID——对所有的 master 线程来说都一样，只要不同线程的表不同就可以了。

我们还说过，线程特定的函数和变量在复制时需要特别对待，但这并不是由服务器处理的。如果语句中引用了服务器变量，这个服务器变量的值将在 slave 上读取。如果你想复制一模一样的值，需要像示例 8-14 那样将该值存储在用户自定义的变量中。或者也可以使用基于行的复制，这个稍后讨论。

◀ 255

## 过滤和跳过事件

有时候事件可能被跳过，因为使用了复制过滤器，或者 slave 明确指示要跳过一些事件。

SQL\_SLAVE\_SKIP\_COUNTER 变量指定 slave 服务器跳过事件的数目。不要在 SQL 线程正在

运行的时候设置该变量。这个条件很容易满足，因为通常需要跳过的事件已经使复制停止了。

当然，导致复制停止的错误需要被审查和处理，但是如果手动解决这个问题，必须忽略停止复制的事件，然后强制复制继续进行。使用这个变量很方便，不再需要 `CHANGE MASTER TO` 命令。示例 8-16 给出了在某个错误语句导致复制停止后使用该变量的例子。

示例8-16：使用 `SQL_SLAVE_SKIP_COUNTER`

```
slave> SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 3;  
Query OK, 0 rows affected (0.02 sec)
```

```
slave> START SLAVE;  
Query OK, 0 rows affected (0.02 sec)
```

在启动 slave 的时候，恢复复制之前需要跳过 3 个事件。如果跳过这 3 个事件会导致某个事务中断，那么在事务执行结束后再继续跳过事件。

如果配置了复制过滤器，还可以由 slave 进行事件过滤。第 4 章中讨论过，master 可以处理过滤，但如果有 slave 过滤器，事件将在 SQL 线程中过滤，即 master 仍然发送事件并存储在中继日志中。

数据库过滤器和表过滤器的过滤过程不同。对某个数据库来说，判断语句是否应该从二进制日志过滤的逻辑在第 4 章中已经详细介绍过。这个逻辑同样适用于 slave 过滤器，不过 slave 过滤器还需要处理一些表过滤器。

重要的是，某一个表上的过滤器将导致与该过滤器有关的整个语句被复制忽略。slave 上过滤语句的逻辑如图 8-5 所示。

表的过滤很容易变得难以理解，为了避免得到不想要的结果，建议遵循以下规则：

256 >

- 不要限定数据表所在的数据库。在语句前面使用 `USE` 语句，设定一个新的默认数据库。
- 不要在单个语句中更新不同数据库中的表。
- 不要在单个语句中更新多个表，除非你知道所有这些表都要过滤或都不会过滤。注意图 8-5 中所示的逻辑是，哪怕只有一个表被过滤，整个语句都将被过滤。



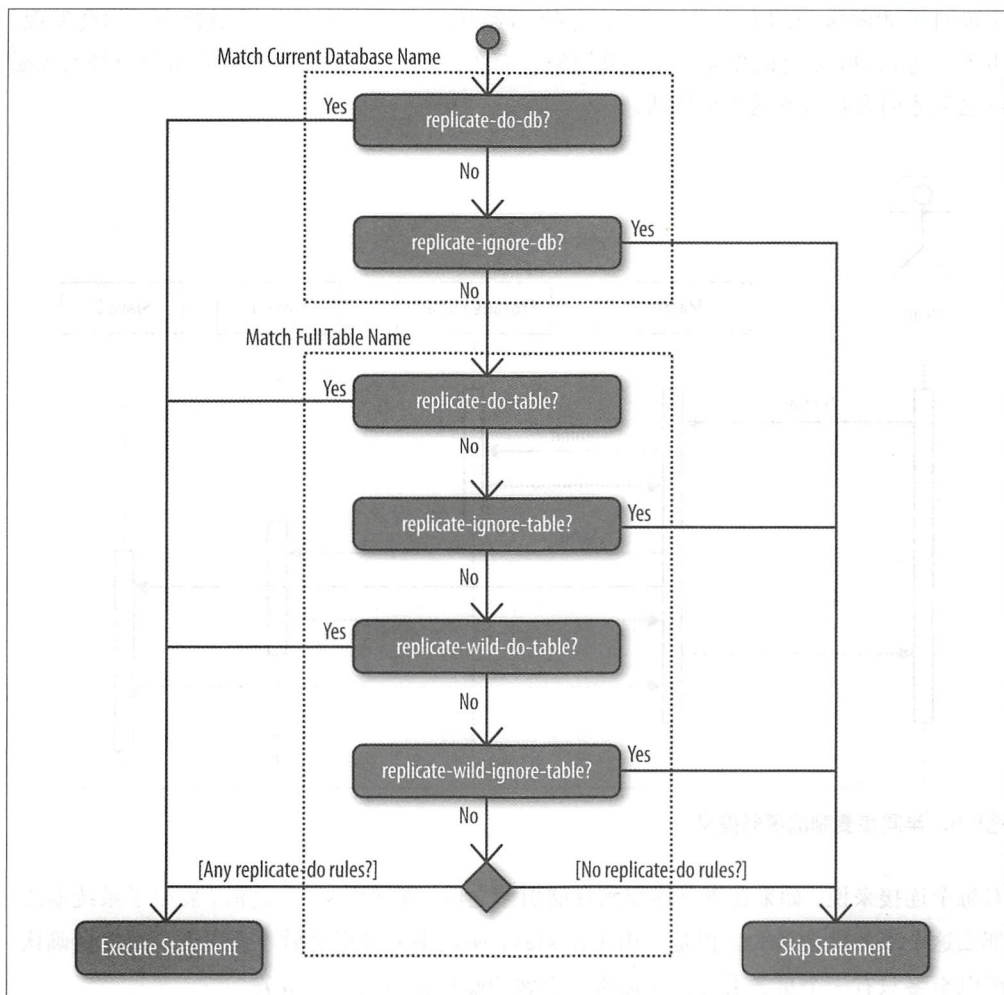


图8-5: 复制过滤规则

## 半同步复制

257

Google 有一组 MySQL 和 InnoDB 的扩展补丁，以适应服务器和存储引擎。其中一个补丁适用于 MySQL 5.0 版本，即半同步复制补丁。MySQL 重做了这个补丁，并随 MySQL 5.5 发布。

半同步复制的原理是在复制继续运行之前，确保至少有一个 slave 将变更写到磁盘。也就是说，对每个连接来说，如果发生 master 崩溃，至多只有一个事务丢失。

一定要理解半同步复制补丁并不会阻止事务的提交，而是直到事务写入至少一个 slave

中继日志才向客户端发送发送响应。图 8-6 给出了提交事务时的调用顺序。你会发现，事务发送到 slave 之前先提交到存储引擎，而在（至少一个）slave 确定事务已经写入磁盘之后才向客户端发送提交确认。

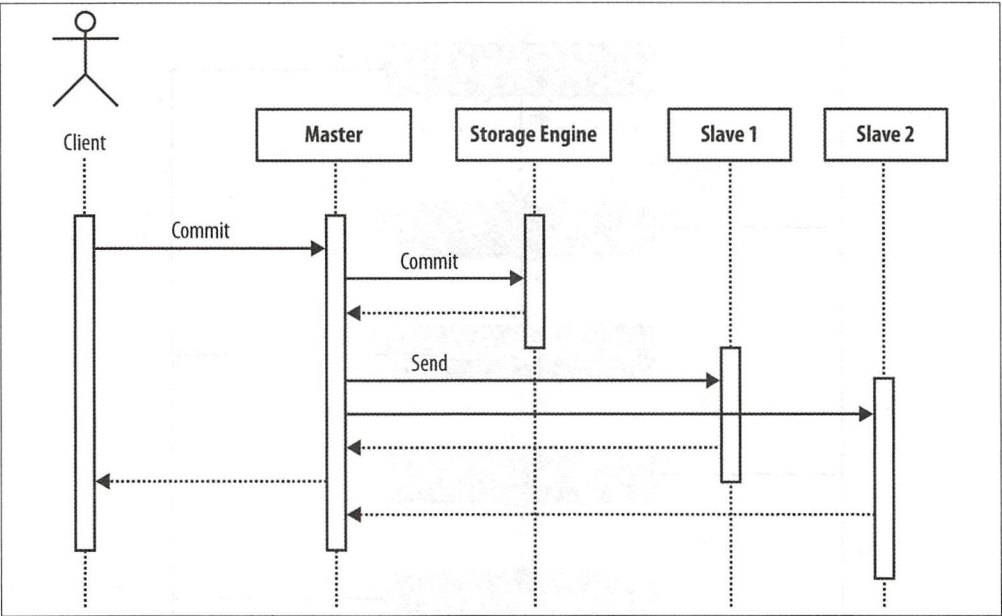


图8-6：半同步复制的事务提交

对每个连接来说，如果在事务提交到存储引擎之后、发送到 slave 之前，发生了系统崩溃，那么这个事务就会丢失。但是，由于在 slave 确定事务提交之后才会向客户端发送确认，所以至多只有一个事务丢失，即通常一个客户端只会丢失一个事务。

## 配置半同步复制

使用半同步复制要求 master 和 slave 都支持，所以 master 和 slave 都必须是 MySQL 5.5 或之后的版本而且启用了半同步复制。如果某一方不支持，那就无法使用半同步复制，当然复制工作还是照常工作，不过如果没有特殊预防措施确保每个事务在新事务启动之前到达 slave 的话，就可能丢失不止一个事务。

启用半同步复制的步骤如下：

1. 在 master 上安装 master 插件：

```
master> INSTALL PLUGIN rpl_semi_sync_master SONAME 'semisync_master.so';
```

## 2. 在每个 slave 上安装 slave 插件：

```
slave> INSTALL PLUGIN rpl_semi_sync_slave SONAME 'semisync_slave.so';
```

## 3. 所有插件安装完毕后，在 master 和 slave 上启用它们。通过两个服务器变量控制的同时也是选项实现，保证设置即使重启也继续生效。最好是关闭服务器，向 master 的 *my.cnf* 文件添加选项：

```
[mysqld]
rpl-semi-sync-master-enabled = 1
```

然后向 slave 的 *my.cnf* 文件添加选项：

```
[mysqld]
rpl-semi-sync-slave-enabled = 1
```

## 4. 重启服务器。

如果按照以上步骤执行，就配置好了半同步复制，测试配置，考虑下面几种情况：

- 如果所有 slave 都崩溃了，无法确认事务是否写入中继日志了怎么办？如果 master 只连接一个 slave，这种情况不是不可能。
- 如果所有 slave 的连接都断了怎么办？这时，master 就无法将事务发出去。

除了 *rpl-semi-sync-master-enabled* 和 *rpl-semi-sync-slave-enabled*，处理以上情况还需要以下两个选项：

*rpl-semi-sync-master-timeout=milliseconds*

259

为了防止半同步复制收不到确认被阻塞，使用 *rpl-semi-sync-master-timeout=milliseconds* 选项进行超时设置。

如果 master 在超时之后仍然收不到任何确认，就还原为常规的异步复制继续操作，不再使用半同步复制。

这个选项也用作服务器变量，不需要停止服务器即可设置。但是要注意，同所有服务器变量一样，一旦重启，服务器变量的值不再有效。

*rpl-semi-sync-master-wait-no-slave={ON|OFF}*

如果事务提交了但 master 没有任何连接的 slave 可用，master 就无法将事务发送出去。默认情况下，master 会等待 slave 连接——在超时限制内——然后确认事务已经正确写入磁盘。

也可以使用 *rpl-semi-sync-master-wait-no-slave={ON|OFF}* 选项关闭这个行为，这时如果没有连接的 slave，master 就还原为异步复制。



注意，如果在 `rpl-semi-sync-master-timeout` 超时之前 `master` 没有收到任何确认，或者如果 `rpl-semi-sync-master-wait-no-slave=ON`，半同步复制都会自动还原为常规的异步复制继续复制操作，不再使用半同步复制。

## 监控半同步复制

这两个插件都有大量状态变量，能够监控半同步复制。这里我们讨论最有趣的几个（全部状态变量参考在线参考手册中半同步复制那部分）：

### `rpl_semi_sync_master_clients`

这个状态变量提供连接在 `master` 上的、支持并启用半同步复制的 `slave` 数目。

### `rpl_semi_sync_master_status`

表示 `master` 上的半同步复制状态，1 表示活动状态，0 表示非活动状态。如果未启用半同步复制，或者启用了但还原成了异步复制，这个值为 0。

### 260 `rpl_semi_sync_slave_status`

表示 `slave` 上的半同步复制状态，1 表示活动状态（比如，如果启用了半同步复制，而且 I/O 线程正在运行），0 表示非活动状态。

使用 `SHOW STATUS` 命令或者信息模式表 `GLOBAL_STATUS`，能够查看这些变量的值。如果想把这些值用作其他用途，`SHOW STATUS` 命令不好用，可以在信息模式表上执行 `SELECT` 查询抽取变量值，然后将其存储到用户自定义变量中，如示例 8-17 所示。

示例 8-17：使用信息模式获取值

```
master> SELECT Variable_value INTO @value
-> FROM INFORMATION_SCHEMA.GLOBAL_STATUS
-> WHERE Variable_name = 'Rpl_semi_sync_master_status';
Query OK, 1 row affected (0.00 sec)
```

## 全局事务标识符

从 MySQL 5.6 开始，引入了全局事务标识符（global transaction identifiers, GTID）的概念，即每个事务都有一个唯一的标识符。这一节介绍 GTID 及其使用。要了解更多 GTID 的细节，请参阅 MySQL 5.6 参考手册中“使用全局事务标识符进行复制”一节（<http://bit.ly/global-ti>）。

在 MySQL 5.6 中，服务器上的每个事务都被分配一个唯一的事务标识符，这是一个 64 位的非零数值，根据事务提交的顺序分配。这个值是服务器本地的（即其他服务器可以为其他事务分配相同的值）。要使事务标识符成为全局的，还要加上服务器的



UUID，构成一对。例如，如果服务器的 UUID（由服务器变量 @@server\_uuid 可得）是 2298677f-c24b-11e2-a68b-0021cc6850ca，那么该服务器上第 1477 个提交的事务的 GTID 是 2298677f-c24b-11e2-a68b-0021cc6850ca:1477。

如果事务从 master 复制到 slave，事务的二进制位置会发生改变，因为 slave 需要将该事务写入 slave 上的二进制日志文件。由于 slave 的配置可能与 master 不同，这个位置可能与 master 位置差别很大，但是全局事务标识符是一样的。

复制事务的时候如果启用了全局事务标识符，不管事务被复制了多少次，事务的 GTID 保持不变。这个简单的想法使 GTID 非常强大，稍后你就会看到这一点。

刚才的标记方法表示单个事务，还需要标记一组全局事务标识符（即 GTID 组）。例如，在提及服务器上记录过哪些事务时，可能会用到。GTID 组定义某个（或一组）范围内的事务标识符。例如，911-1066 和 1477-1593 的一组事务记为 2298677f-c24b-11e2-a68b-0021cc6850ca:911-1066:1477-1593。

261



GTID 被写入二进制日志，并且只会分配给已经写入二进制日志的事务。也就是说，如果关闭二进制日志，事务就不会分配 GTID 了。不管 master 还是 slave 都是这样。所以，如果想使用 slave 做故障转移，需要开启它的二进制日志。如果没有开启二进制日志，slave 就不会记下事务的 GTID。

## 使用 GTID 配置复制

使用全局事务标识符配置复制，必须在配置服务器的时候启用全局事务标识符。这里我们介绍一下启用全局事务标识符的步骤。要使用全局事务标识符配置备用服务器，需要按照以下方式更改 *my.cnf* 文件：

```
[mysqld]
user      = mysql
pid-file  = /var/run/mysqld/mysqld.pid
socket    = /var/run/mysqld/mysqld.sock
port      = 3306
basedir   = /usr
datadir   = /var/lib/mysql
tmpdir    = /tmp
log-bin    = master-bin ❶
log-bin-index = master-bin.index
server-id  = 1
gtid-mode  = ON ❷
log-slave-updates ❸
enforce-gtid-consistency ❹
```

- ① 备用服务器上必须启用了二进制日志。当 master 成为主服务器的时候，确保所有变更都会被写入二进制日志，不过还需要 `log-slave-updates`。
- ② 该选项用来生成全局事务标识符。
- ③ 该选项保证来自 master 的已执行的事件同样也会写入备用服务器的二进制日志。否则，备用服务器就无法将变更间接发给 slave。注意，默认情况下选项是未启用的。
- ④ 该选项确保如果语句的记录与全局事务标识符不一致，语句就会报错。为了正确处理故障转移，推荐使用这个选项。

262 ➤ 更新完配置文件以后，需要重启服务器，使改变生效。所有服务器都做了这些配置以后，就可以做故障转移了。在 MySQL 5.6 中，使用 GTID 切换 master 只需要下面的命令就可以了：

```
CHANGE MASTER TO
MASTER_HOST = host_of_new_master,
MASTER_PORT = port_of_new_master,
MASTER_USER = replication_user_name,
MASTER_PASSWORD = replication_user_password,
MASTER_AUTO_POSITION = 1
```

MASTER\_AUTO\_POSITION 使 slave 在连接 master 的时候，自动与 master 协商应该发送什么事务。

使用 SHOW SLAVE STATUS 查看复制的 GTID 位置，如示例 8-18 所示，这时输出多了几个列：

示例8-18：启用GTID时SHOW SLAVE STATUS的输出

```
Slave_IO_State: Waiting for master to send event
.
.
.
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
.
.
.
Master_UUID: 4e2018fc-c691-11e2-8c5a-0021cc6850ca
.
.
.
Retrieved_Gtid_Set: 4e2018fc-c691-11e2-8c5a-0021cc6850ca:1-1477
Executed_Gtid_Set: 4e2018fc-c691-11e2-8c5a-0021cc6850ca:1-1593
Auto_Position: 1
```

## Master\_UUID

这是 master 的 UUID，不一定与 GTID 的实现有关（其实在引入 GTID 之前就有这个了）。在排除故障时非常有用。

## Retrieved\_Gtid\_Set

这是从 master 获取而来的、存储在中继日志中的一组 GTID。

## Executed\_Gtid\_Set

这是 slave 上已经执行、并且已经写入 slave 的二进制日志的一组 GTID。

# 使用 GTID 进行故障转移

263

第 5 章的“热备份”一节讲过如何在不使用全局事务标识符的情况下切换到热备份，其中需要使用二进制日志位置。而使用全局事务标识符时，不再需要检查这些位置。

使用全局事务标识符切换到热备份很简单（只需要使用 `CHANGE MASTER` 将 slave 重定向到新的 master 即可）：

```
CHANGE MASTER TO MASTER_HOST = 'standby.example.com';
```

如果其他参数没变，没必要重新再写一遍。

启用 `MASTER_AUTO_POSITION` 时，master 会搞清楚需要发送哪些事务。因此，使用 `Replicant` 库，很容易定义故障转移过程：

```
_CHANGE_MASTER = (  
    "CHANGE MASTER TO "  
    "MASTER_HOST = %s, MASTER_PORT = %d, "  
    "MASTER_USER = %s, MASTER_PASSWORD = %s, "  
    "MASTER_AUTO_POSITION = 1"  
)
```

```
def change_master(server, master):  
    server.sql(_CHANGE_MASTER,  
               master.host, master.port,  
               master.user, master.password)
```

```
def switch_to_master(server, standby):  
    change_master(server, standby)  
    server.sql("START SLAVE")
```

对比示例 5-1 和这个方法，你会发现有些东西能够通过 GTID 进行改进：

- 由于不必再检查 master 上的位置了，所以也就不需要停止 master 确保位置不变。
- 由于 GTID 是全局的（即在复制过程中保持不变），slave 没必要与 master 位置一致，备用服务器也没必要等待一个好的切换位置。
- 不需要获取备用服务器（其实是当前 master 的一个 slave）位置，因为所有东西都会复制到 slave 上。
- 更改 master 时不需要提供位置，因为服务器会自动协商这些位置。

因为 GTID 是全局的（即不需要对位置做任何解释翻译），所以上面的方法能够完成切换和故障转移，哪怕在级联复制结构中也适用。这就不同于第 5 章的“热备份”一节，那里处理切换、非级联故障转移和级联故障转移，分别要采用不同的方法。

为了避免 master 失效时丢失事务，要养成执行故障转移之前清空中继日志的好习惯。这样就避免了重复从 master 获取那些已经发送到 slave 的事务。最好是只将 I/O 线程重定向到新的 master，但不幸的是，这还做不到。为了等待中继日志变空，使用 `WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS` 函数阻塞直到 GTID 组中的所有 GTID 都被 SQL 线程处理完毕。这个函数的使用见示例 8-19。

示例8-19：使用GTID将故障转移到备用服务器的Python代码

```
def change_master(server, standby):
    fields = server.sql("SHOW SLAVE STATUS")
    server.sql("SELECT WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS(%s)",
               fields['Retrieved_Gtid_Set'])
    server.sql("STOP SLAVE")
    change_master(server, standby)
    server.sql("START SLAVE")
```

## 使用 GTID 提升 slave

如果 slave 确实比备用服务器滞后，那么上一节介绍的故障转移方法就没问题。但是，第 5 章的“提升 slave”一节中提过，如果 slave 上的事务比备用服务器还多，将故障转移到备用服务器就不能解决问题。这时，如果 slave 是新的 master 就更好了。那么，怎么通过全局事务标识符来实现呢？

示例 8-19 中的故障转移方法还是有用的，但是如果一个 master 有多个 slave，而 master 失效了，就需要比较这些 slave，看看谁“知道的比较多”。为此，MySQL 5.6 引入了一个变量 `GTID_EXECUTED`。这个全局变量包含一个 GTID 组，其中的所有事务都已经被写入了二进制日志。注意，只有当事务写入二进制日志以后才会产生 GTID，所以只有写入二进制日志的事务才会出现在 GTID 组中。



还有一个全局变量 `GTID_PURGED`，包含已经从二进制日志清除（即删除）从而复制不可用的所有事务的集合。这个集合总是 `GTID_EXECUTED` 的子集（或相同）。

这个变量用来检查每个候选 master 的二进制日志上是否有足够的事务，使它能够担当 master 的角色。如果 master 上的 `GTID_PURGED` 里面有事件，而这些事件不在 slave 的 `GTID_EXECUTED` 中，master 就不能把这些需要的事件复制到 slave，因为这些事件不在二进制日志中。两个变量的关系如图 8-7 所示，其中每个变量用所有 GTID 空间上的一个波面表示。

265

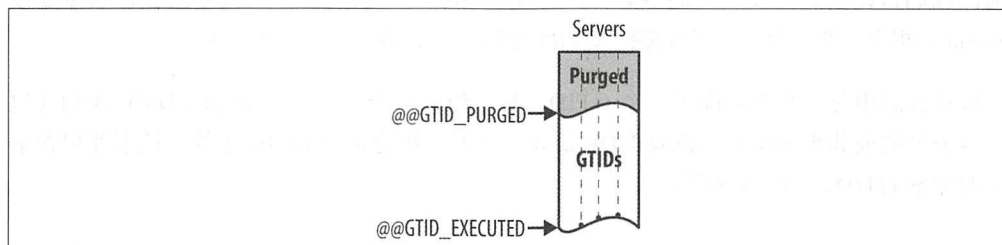


图8-7: `GTID_EXECUTED`和`GTID_PURGED`

使用 `GTID_EXECUTED` 很容易比较 slave，然后决定哪个 slave “知道最多” 事务。示例 8-20 基于 `GTID_EXECUTED` 对 slave 进行排序，然后选择 “最好的” 那个作为新的 master。注意，GTID 组通常不是有序的（即两个相同大小的 GTID 组可能不一样）。但是在这个例子中，GTID 组是完全有序的，它们是在 master 的二进制日志中被排序的。

示例8-20: 寻找最佳slave的Python代码

```
from mysql.replicant.server import GTIDSet
```

```
def fetch_gtid_executed(server):
    return GTIDSet(server.sql("SELECT @@GLOBAL.GTID_EXECUTED"))
```

```
def fetch_gtid_purged(server):
    return GTIDSet(server.sql("SELECT @@GLOBAL.GTID_PURGED"))
```

```
def order_slaves_on_gtid(slaves):
    entries = []
    for slave in slaves:
        pos = fetch_gtid_executed(slave)
        entries.append((pos, slave))
    entries.sort(key=lambda x: x[0])
    return entries
```

合并示例 8-19 和示例 8-20 中的代码，使函数能够将最佳 slave 提升为 master，如示例 8-21 所示。

示例8-21: MySQL 5.6使用GTID提升slave

```
def promote_best_slave_gtid(slaves):
    entries = order_slaves_on_gtid(slaves)
    _, master = entries.pop(0)          # “最佳” slave 将作为新的 master
    for _, slave in entries:
        switch_to_master(master, slave)
```

## 266 GTID 的复制

前面我们讨论了如何使用全局事务标识符配置 MySQL 服务器, 以及如何进行故障处理和 slave 提升, 但是还有一个问题: GTID 是如何在服务器之间复制的呢?

二进制日志中每个组都分配了一个 GTID, 每个组即每个事务、单语句的 DML 语句(包括事务型的或非事务型的)以及 DDL 语句。在写入组之前写 GTID 事件, 该事件包含事务的完整 GTID, 如图 8-8 所示。

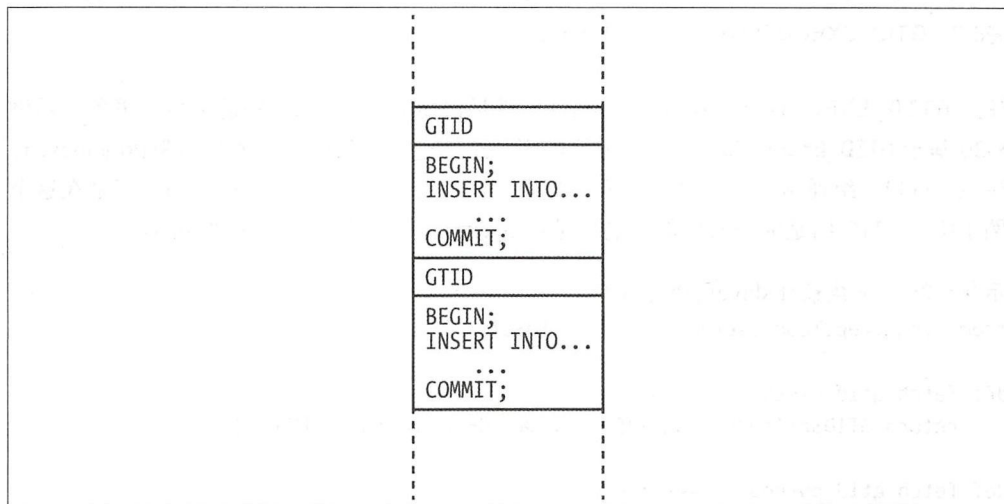


图8-8: 包含GTID的二进制日志文件

为了处理带有 GTID 的事务的复制, SQL 线程按照以下方式处理 GTID 事件:

1. 如果 GTID 已经在 GTID\_EXECUTED 中, 跳过整个事务, 也不写入二进制日志。(回忆一下, GTID\_EXECUTED 包含所有已经写入二进制日志的事务, 所以没必要再写一次。)
2. 否则, GTID 就被分配给后面的事务, 下一个事务正常执行。
3. 如果事务提交, 事务的 GTID 就用来产生一个新的 GTID 事件, 然后这个事件在事务之前写入二进制日志。

4. 然后在 GTID 事件之后，事务缓存的内容被写入二进制日志。

注意，由于每个事务都分配了 GTID，所以可以在第一步过滤掉那些已经执行过的事务，而 MySQL 5.6 之前的版本则做不到这一点。

通过一个新的变量 `GTID_NEXT` 能够控制分配什么 GTID 给事务。这个变量包含一个 GTID 或 `AUTOMATIC` 值。（也可以取 `ANONYMOUS` 的值，但只能在 `GTID_MODE = ON` 时使用，所以这里不考虑这种情况。）当提交事务的时候，根据 `GTID_NEXT` 的值有不同的操作：

- 如果 `GTID_NEXT` 的值为 `AUTOMATIC`，那么创建一个新的 GTID 并将其分配给事务。
- 如果 `GTID_NEXT` 的值为某个 GTID，那么使用这个 GTID 并且会随事务一起写入二进制日志。

事务提交之后，`GTID_NEXT` 中的 GTID 不变。所以，提交事务之后，必须设置新的 GTID 或者使用 `AUTOMATIC`。如果不更改 `GTID_NEXT` 的值，在尝试开始新事务的时候就会出错，不管新事务是显式的还是隐式的。

可以发现，在事务开始之前就要设置 `GTID_NEXT`。如果尝试在开始事务之后设置变量，就会报错。

设置了 `GTID_NEXT` 并且开始了新事务，GTID 就被这个事务拥有了，由变量 `GTID_OWNED` 体现：

```
mysql> SELECT @@GLOBAL.GTID_OWNED;
+-----+
| @@GLOBAL.GTID_OWNED |
+-----+
| 02020202-0202-0202-0202-020202020202:4#42 |
+-----+
1 row in set (0.00 sec)
```

在这个例子中，只有一个 GTID 被拥有，即 `02020202-0202-0202-0202-020202020202:4`，其拥有者是 ID 为 42 的会话。

`GTID_OWNED` 应该是内部变量，用于测试和调试。

直接从 master 到 slave 的复制并不是复制变更的唯一途径。MySQL 复制还可以使用 `mysqlbinlog`，生成所有 SQL 语句，保存到文件，然后应用到服务器。不管复制是不是通过 `mysqlbinlog` 间接完成的，都使用 `GTID_NEXT` 处理 GTID 的复制。每当 `mysqlbinlog` 遇到一个 GTID 事件，就产生一条语句设置 `GTID_NEXT` 的值。示例 8-22 给出了一个输出示例。

**268** 示例8-22: 带有GTID事件的mysqlbinlog输出示例

```
# at 410
#130603 20:57:54 server id 1 end_log_pos 458 CRC32 0xc6f8a5eb
# GTID [commit=yes]
SET @@SESSION.GTID_NEXT= '01010101-0101-0101-0101-010101010101:3'/*!*/;
# at 458
#130603 20:57:54 server id 1 end_log_pos 537 CRC32 0x1e2e40d0
# Position Timestamp Type Master ID Size Master Pos Flags
# Query hread_id=4 exec_time=0 error_code=0
SET TIMESTAMP=1370285874/*!*/;
BEGIN
/*!*/;
# at 537
#130603 20:57:54 server id 1 end_log_pos 638 CRC32 0xc16f211d
# Query thread_id=4 exec_time=0 error_code=0
SET TIMESTAMP=1370285874/*!*/;
INSERT INTO t VALUES (1004)
/*!*/;
# at 638
#130603 20:57:54 server id 1 end_log_pos 669 CRC32 0x91980f0b
COMMIT/*!*/;
```

## slave 的安全和恢复

slave 服务器也有可能崩溃, 如果 slave 崩溃, 必须进行恢复。处理 slave 崩溃的第一步是搞清楚崩溃的原因。这个过程无法自动完成, 因为崩溃的原因很多且无法预测。可能是磁盘容量不足, 读取损坏事件, 或者由于某种原因重新执行语句, 导致重复关键字错误等。但是, 某些恢复过程可以自动执行, 而且可以自动诊断错误。

## 同步、事务以及数据库崩溃问题

为了保证 master 或 slave 崩溃以后 slave 能够安全地恢复复制, 需要考虑以下两个方面的问题:

- 保证 slave 上存有恢复所需的所有数据。
- 执行 slave 的恢复。

slave 通过磁盘同步尽量满足第一个条件。操作系统将所需的文件存在内存中, 并周期性地(或强制)写入磁盘, 从而提供了可接受的性能。也就是说, 写入文件的数据不一定安全, 一旦发生崩溃, 内存中的数据将丢失。

**269** 为了强制 slave 将文件写入磁盘, 数据库服务器发出 `fsync` 调用, 将所有内存中的数据



写入磁盘。为了保护复制数据，通常 MySQL 服务器定期在中继日志、*master.info* 文件和 *relay-log.info* 文件上执行 `fsync` 调用。

## I/O 线程同步

对于 I/O 线程来说，无论什么时候处理事件都有两个 `fsync` 调用：一个将中继日志刷新 (`flush`) 到磁盘，另一个将 *master.info* 文件刷新到磁盘。这种刷新顺序保证了没有事件丢失，即使 slave 在刷新中继日志和刷新 *master.info* 文件之间的某个时候崩溃。但是，如果在以下几种情况发生崩溃，可能产生重复事件：

- 服务器刷新中继日志，正要更新 *master.info* 文件中的 master 读位置。
- 服务器崩溃，也就是说，master 读位置现在指向事件被刷新到中继日志之前的位置。
- 服务器重启，并从 *master.info* 获得 master 读位置，即在最后一个事件写入中继日志之前的位置。
- 从这个位置恢复复制，导致事件重复。

如果按相反顺序刷新文件（先 *master.info* 文件再中继日志），可能丢失事件，因为 slave 会在事件之后恢复复制。我们认为丢失事件比重复事件严重，因此要先刷新中继日志。

## SQL 线程同步

SQL 线程轮流处理每个事件来处理中继日志中的组。在处理组中的所有事件时，SQL 线程按以下方式提交事务：

1. 将事务提交到存储引擎（假定存储引擎支持提交）。
2. 更新 *relay-log.info* 文件的下一个事件的位置，这个位置也是处理下一个组的开始位置。
3. 发出 `fsync` 调用，将 *relay-log.info* 文件写入磁盘。

在组内执行时，通过增加事件的位置，跟踪 SQL 线程对中继日志的读取位置。但是如果发生崩溃，将从 *relay-log.info* 文件的最后一个记录位置恢复执行。

这种方式使得 SQL 线程的原子更新问题与前面的 I/O 线程不同，所以下面的几种情况可能导致 slave 数据库和 *relay-log.info* 文件不同步： ◀ 270

1. 事件在数据库上已应用，且事务已提交。下一步是更新 *relay-log.info* 文件。
2. slave 崩溃，也就是说，*relay-log.info* 文件现在指向刚刚完成的事务的开始。
3. 恢复时，SQL 线程从 *relay-log.info* 文件读取信息，并从保存的位置开始复制。

#### 4. 重复上一次执行的事务。

以上这些情况都是因为，在 slave 上提交事务和更新复制信息并不是原子性操作：*relay-log.info* 文件并不能准确地反映数据库上提交的事务。下一节我们将讨论 MySQL 5.6 中如何通过事务型复制解决这个问题。

## 事务型复制

上一节提到，复制不是崩溃安全的（crash-safe）的，因为复制进度信息并不总是与数据库中的应用实际变更同步。即使服务器崩溃时事务没有丢失，也需要花点力气将 slave 恢复回来。

MySQL 5.6 通过提交事务时同时提交复制信息，增强了 slave 的崩溃安全性，如图 8-9 所示。也就是说，复制信息总是与数据库的变更应用一致，不论服务器是否发生崩溃。而且，master 也做了调整保证能够正确恢复。

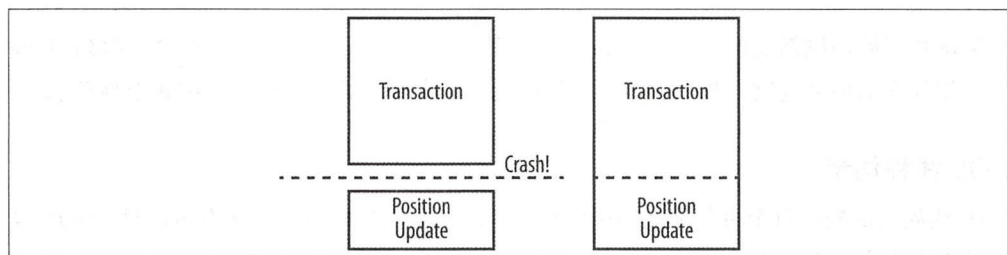


图8-9：事务之后或事务内部的位置信息改变

271 回想一下，复制信息存储在两个文件中：*master.info* 和 *relay-log.info*，它们在事务应用之后更新。所以，如果崩溃发生在事务提交和文件更新之间的某个时候，如图 8-9 的左图，那么位置信息就是错误的。也就是说，事务不会丢失，但是在 slave 恢复的时候，事务可能会再次被应用。

通常避免这个问题的解决办法是，在所有表上都加上主键。这样，重复更新表就会导致 slave 停机，使用 `SQL_SLAVE_SKIP_COUNTER` 能够跳过事务然后让 slave 重新运行（或使用 `GTID_NEXT` 提交一个假事务）。这个方法肯定比丢失事务好多了，但也是个麻烦精。通过删除主键来防止 slave 停机只能解决一部分问题：事务会被应用两次，这会给应用程序带来负担即要处理两条记录，同时还要求定期清理表。这两件事都需要人工干预或者脚本支持。这不会影响稳定性，但是如果在提交事务的时候同时提交了复制信息，崩溃就更容易处理。

MySQL 5.6 要实现事务型复制，将复制信息存储在文件（就像前面那样）或者表中。即使复制信息存储在表中，数据和复制信息要么使用相同的存储引擎（必须是事务型存储引擎），要么两个存储引擎都要支持 XA。如果两个条件都不满足，复制信息和数据就无法以单个事务提交。

## 配置事务型复制

MySQL 5.6 默认使用文件存储复制信息，所以需要将服务器重新配置成使用表存储复制信息，才能使用事务型复制。使用两个新选项控制复制信息的存放位置：`master_info_repository` 和 `relay_log_info_repository`。这两个选项的值为 `FILE` 或 `TABLE`，分别表示使用文件或表存储复制信息。

因此，为了使用事务型复制，编辑配置文件，添加选项，像示例 8-23 那样，然后重启服务器。

示例8-23：添加选项开启事务型复制

```
[mysqld]
...
master_info_repository = TABLE
relay_log_info_repository = TABLE
...
```



MySQL 5.6.6 之前，`slave_master_info` 和 `slave_relay_log_info` 的默认引擎是 `MyISAM`。为了使用事务型复制，需要将引擎改成事务型引擎，比如 `InnoDB`。使用 `ALTER TABLE` 命令实现：

```
slave> ALTER TABLE mysql.slave_master_info ENGINE = InnoDB;
slave> ALTER TABLE mysql.slave_relay_log_info ENGINE = InnoDB;
```

◀ 272

## 事务型复制的细节

`mysql` 数据库中保存事务型复制相关信息的两张表分别是：1) `slave_master_info`，对应 `master.info` 文件；和 2) `slave_relay_log_info`，对应 `relay_log.info` 文件。

像 `master.info` 文件那样，`slave_master_info` 表保存 `master` 的信息。表 8-1 列出了这个表的各个字段，以及每个字段对应 `master.info` 文件中的行及 `SHOW SLAVE STATUS` 命令的输出。

表8-1：slave\_master\_info表中的字段

字段	文件中的行	slave 状态列
Number_of_lines	1	
Master_log_name	2	Master_Log_File

续表

字段	文件中的行	slave 状态列
Master_log_pos	3	Read_Master_Log_Pos
Host	3	Master_Host
User_name	4	Master_User
User_password	5	
Port	6	Master_Port
Connect_retry	7	Connect_Retry
Enabled_ssl	8	Master_SSL_Allowed
Ssl_ca	9	Master_SSL_CA_File
Ssl_capath	10	Master_SSL_CA_Path
Ssl_cert	11	Master_SSL_Cert
Ssl_cipher	12	Master_SSL_Cipher
Ssl_key	13	Master_SSL_Key
Ssl_verify_servert_cert	14	Master_SSL_Verify_Server_Cert
Heartbeat	15	
Bind	16	Master_Bind
Ignored_server_ids	17	Replicate_Ignore_Server_Ids
Uuid	18	Master_UUID
Retry_count	19	Master_Retry_Count
Ssl_crl	20	Master_SSL_Crl
Ssl_crlpath	21	Master_SSL_Crlpath
Enabled_auto_position	22	Auto_Position

同样，表 8-2 列出了 slave\_relay\_log\_info 表中的字段及其对应 relay\_log.info 文件中的行。

表8-2: slave\_relay\_log\_info表中的字段

字段	文件中的行	slave 状态列
Number_of_lines	1	
Relay_log_name	2	Relay_Log_File
Relay_log_pos	3	Relay_Log_Pos
Master_log_name	4	Relay_Master_Log_File
Master_log_pos	5	Exec_Master_Log_Pos
Sql_delay	6	SQL_Delay
Number_of_workers	7	
Id	8	

现在，假定在 master 上执行下面这些事务：

```
START TRANSACTION;
UPDATE titles, employees SET titles.title = 'Dictator-for-Life'
  WHERE first_name = 'Calvin' AND last_name IS NULL;
UPDATE salaries SET salaries.salary = 1000000
  WHERE first_name = 'Calvin' AND last_name IS NULL;
COMMIT;
```



当事务到达 slave 然后在 slave 上执行的时候，它的执行其实是这样的（其中 Exec\_Master\_Log\_Pos、Relay\_Master\_Log\_File、Relay\_Log\_File 和 Relay\_Log\_Pos 来自于 SHOW SLAVE STATUS 的输出）：

```
START TRANSACTION;
UPDATE titles, employees SET titles.title = 'Dictator-for-Life'
  WHERE first_name = 'Calvin' AND last_name IS NULL;
UPDATE salaries SET salaries.salary = 1000000
  WHERE first_name = 'Calvin' AND last_name IS NULL;
SET @@SESSION.LOG_BIN = 0;
UPDATE mysql.slave_relay_log_info
  SET Master_log_pos = Exec_Master_Log_Pos,
      Master_log_name = Relay_Master_Log_File,
      Relay_log_name = Relay_Log_File,
      Relay_log_pos = Relay_Log_Pos;
SET @@SESSION.LOG_BIN = 1;
COMMIT;
```

注意，加进来的“语句”并不会写入 master 的二进制日志，因为在这个“语句”执行的时候二进制日志处于暂时禁用的状态。如果 slave\_relay\_log\_info 表和所有表都使用同一个引擎，这将作为一个整体提交。

274

结果是 slave 上执行的每一个事务都被更新到 slave\_relay\_log\_info 表。但是注意，slave\_master\_info 表不包含那些确保事务型复制正常运行的重要信息，只保存从 master 上获取的事件的位置。如果发生崩溃，slave 将从上一次执行的位置恢复，而不是上一次获取的位置恢复，所以这个信息只对 master 崩溃有用。这时，中继日志中的事件将被执行，避免更多事件丢失。

同刷新（flush）到磁盘一样，提交到表同样开销较大。因为 slave\_master\_info 表不包含确保事务型复制正常运行的任何重要信息，避免在这个表上进行不必要的提交，能够提高性能。

因此，引入 sync\_master\_info 选项。这个选项是一个整数，表示复制信息应该被提交到 slave\_master\_info 表（或者刷新到磁盘，当信息存储在文件中的时候）的频率。如果这个值非零，每次 master 获取到的事件数目等于这个值的时候，刷新复制信息。如果这个值是 0，没有任何显式刷新操作，但是操作系统会将信息刷新到磁盘。注意，当二进制日志轮换或者 slave 启动和停止的时候，信息都会被刷新到磁盘或提交到表。

如果使用表存储复制信息，即：

```
sync_master_info = 0
```

那么只有当 slave 启动和停止或者二进制日志轮换时，slave\_master\_info 表才被更新，所以其他线程看不到已获取的位置的改变。如果应用程序需要看到这个信息，就需要将 sync\_master\_info 设置为非零值。

## 保护非事务型语句的规则

崩溃后重新执行时不会跟踪和保护事务之外的语句。这在 master 和 slave 上情况差不多。如果 MyISAM 表上的语句由于 master 崩溃而中断，这个语句将不再记入日志，因为只有执行完成的语句才会写入日志。重启（并修复成功）时 MyISAM 表包含一部分更新，但是二进制日志根本没有记录该语句。

slave 上的情形类似：如果语句（或更改非事务型表的事务）在执行过程中发生崩溃，表的变更可能还在，但是组的位置不变。当 slave 再次启动复制时将重新执行这个非事务型语句。

**275** 在更新非事务型表的过程中，很难自动发现导致崩溃的原因。但是通过观察一些规则，可以保证出现问题时至少能接收到错误信息。

### INSERT 语句

要复制的表中必须有主键。这样，重新执行 INSERT 语句将会产生一个重复键错误，导致 slave 停止，你可以检查一下 master 和 slave 不一致的原因。

### DELETE 语句

避免使用 LIMIT 从句。如果不用 LIMIT 从句，语句将删除相同的行（即匹配 WHERE 从句的那些行），要么从上一个语句遗留的地方继续，要么如果所有行都被删除了则什么都不做。但是，如果语句中有 LIMIT 从句，则只删除符合 WHERE 条件的行的子集。如果再次执行同样的语句，将删除另一个行子集。

### UPDATE 语句

这种语句最复杂。语句必须是幂等的（即两次执行得到同样的结果），或者语句两次执行结果的偶然性是可接受的，比如用于维护页面访问统计数据的 UPDATE 语句。

## 多源复制

你可能注意到了，一个 slave 连接多个 master 并接收来自所有 master 的变更，是不可能的。

这种拓扑称为多源（multisource），不要与第 6 章中的多主（multimaster）拓扑混淆。在多源拓扑中，变更来自多个 master；而在多主拓扑中，将每个 master 上的变更复制到其他所有 master，整个服务器组扮演单个 master 的角色。

将多源复制引入 MySQL 计划已久，但设计上有个问题：如何处理更新冲突。不同的源会带来更新冲突，或者两个中继服务器同时转发了某个 master 的变化，都会产生冲突。图 8-10 解释了这两种冲突。第一种，两个 master（源）同时改变同样的数据，slave 无法判断数据最终如何变化。第二种，只有一个 master 改变，但 slave 接收到两个改变。无论哪种情况，slave 都无法区分两个中继服务器的事件，所以当 master 的事件到达 slave 时，slave 认为收到了两个不同的事件。

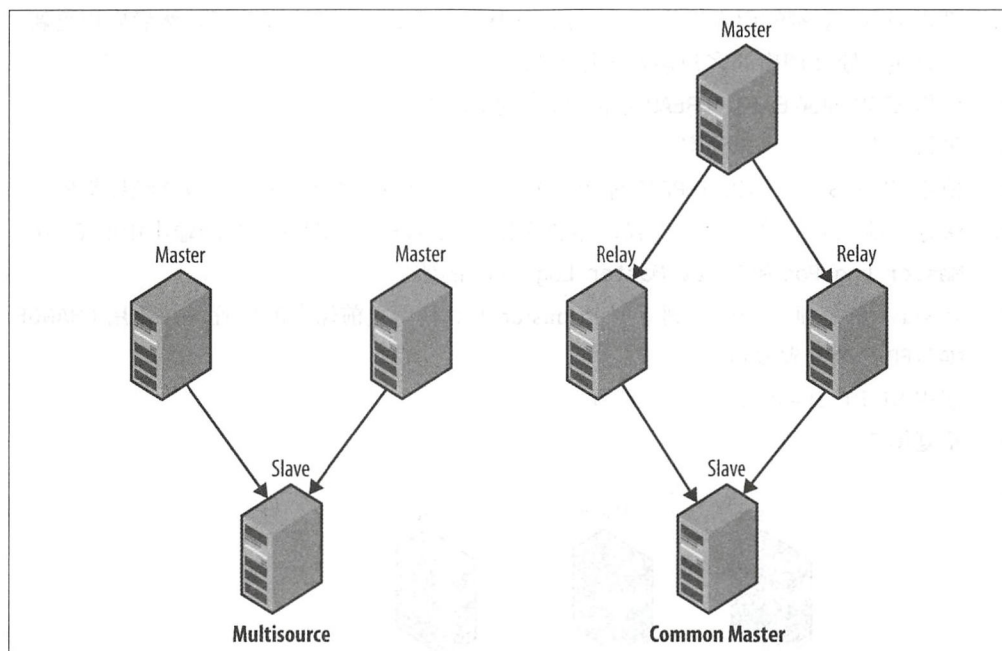


图8-10：真多源和菱形结构



这种菱形结构不需要配置：如果切换过程中复制流重叠，那么从一个中继服务器切换到另一个中继服务器时会不经意间发生这种情况。所以，要保证队列中的所有事件——包括 slave 上以及 master 和 slave 之间的所有中继服务器上的所有事件，在切换到另一个 master 之前已经复制到 slave 上了。

276

正确处理切换可以避免冲突（在多个数据源的情况下），确保所有更新已经完成，从而没有发生冲突的可能。典型的实现方法是，更新不同的数据库，但也可以将更新同一张

表的不同行的操作分配到不同的服务器执行。

尽管目前 MySQL 不支持同时从多个源复制，但可以近似实现：将 slave 在多个 master 之间切换，轮流从其中一个 master 定期复制，称为轮盘多源复制（round-robin multisource replication）。对某些应用来说这种方法很有用，比如从多个源聚集数据做报表的时候。这时，每个 master 的写操作处理各自的数据库、表或分区，自然地将数据分离。由于没有冲突的风险，所以可使用多源复制。

277 图 8-11 给出了 slave 以轮盘的方式从三个 master 复制的例子，有一个客户端专门处理 master 之间的切换。轮盘多源复制的过程如下：

1. 将 slave 配置为从一个 master 进行复制，我们称这个 master 为当前 master。
2. 设置 slave 复制的固定工作时间。slave 从当前 master 中读取更新，然后应用更新，这时负责处理切换的客户端处于休眠状态。
3. 使用 `STOP SLAVE IO_THREAD` 停止 slave 的 I/O 线程。
4. 等待，直到中继日志为空。
5. 使用 `STOP SLAVE SQL_THREAD` 停止 SQL 线程。`CHANGE MASTER` 要求两个线程都停止。
6. 保存当前 master 的 slave 位置，存储 `SHOW SLAVE STATUS` 命令的输出中的 `Exec_Master_Log_Pos` 和 `Relay_Master_Log_File` 的值。
7. 将 slave 的复制按顺序换到下一个 master 上：利用之前保存的位置，并使用 `CHANGE MASTER` 命令配置复制。
8. 使用 `START SLAVE` 重启 slave 线程。
9. 重复第 2 ~ 8 步。

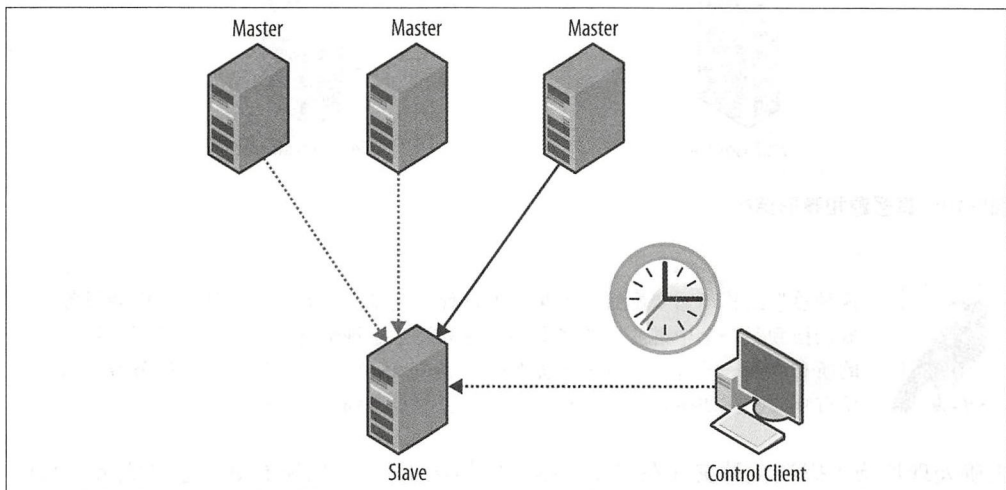


图8-11：轮盘多源复制，由客户端负责切换



注意第 3 ~ 5 步，我们先停止 I/O 线程，然后停止 SQL 线程。之所以要停止这两个线程，而不是仅仅停止 slave 上的复制，原因是 SQL 线程可能拖慢进度（通常都会），所以，如果我们停止了这两个线程，则中继日志中会有大量事件被丢弃。例如，如果你更加关心每个 master 一分钟内执行了多少事务，而不关心丢弃掉的这些事件，那么你只需要停止复制即可，而不用执行第 3 ~ 5 步。但是，这个过程是没有问题的，因为丢弃的事件将会在下一轮从 master 重新读取。

当然，这个过程可以自动化，使用一个单独的客户端连接和 MySQL Replicant 库，如示例 8-24 所示。使用 `itertools` 模块的 `cycle` 函数，可以轮流从一组 master 中反复读取数据。

示例 8-24：轮盘多源复制的 Python 脚本

```
import itertools

position = {}

def round_robin_multi_master(slave, masters):
    current = masters[0]
    for master in itertools.cycle(masters):
        slave.sql("STOP SLAVE IO_THREAD");
        slave_wait_for_empty_relay_log(slave)
        slave.sql("STOP SLAVE SQL_THREAD");
        position[current.name] = fetch_slave_position(slave)
        slave.change_master(position[current.name])
        master.sql("START SLAVE")
        current = master
        sleep(60)                # Sleep 1 minute
```

## 基于行的复制的细节

第 4 章“基于行的复制”一节遗漏了一个基于行的复制的重要问题：这些行是如何在 slave 上执行的。这一小节我们讨论基于行的复制在 slave 端是如何实现的。

在基于语句的复制中，通过将语句写入某个 Query 事件来处理语句。但是，由于每个语句可能更改了大量的行，基于行的复制的处理方法不同，每个语句需要多个事件。

引入 4 个新的事件处理基于行的复制：

### Table\_map

将表 ID 映射为表名（包括数据库名），以及关于 master 上的表的列的基本信息。

表信息不包括列名，只有列的类型。因为基于行的复制是位置性的，即 master 上的每个列按照相同位置被复制到 slave 表。

在插入、删除和更新行的时候分别生成这些事件。也就是说，单个语句能够产生多个事件。除了行以外，每个事件还有一个表ID，这个表ID来自上一个Table\_map事件，还有一个或两个列位图，说明该事件影响表中的哪些列。这样就只需要记录那些发生变动的列，或定位插入、删除或更新的行需要的列，从而节省了空间。

执行语句的时候，会以一组Table\_map事件序列及一组行事件序列的形式，将语句写入二进制日志。语句的最后一个行事件有一个特殊的标记，表明这是语句的最后一个事件。

示例 8-25 给出了语句的执行及其产生的事件。这里我们忽略了格式描述事件，因为前面已经讲过了。

示例8-25: INSERT语句的执行及其事件

```
master> START TRANSACTION;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
master> INSERT INTO t1 VALUES (1),(2),(3),(4);
```

```
Query OK, 4 rows affected (0.01 sec)
```

```
Records: 4 Duplicates: 0 Warnings: 0
```

```
master> INSERT INTO t1 VALUES (5),(6),(7),(8);
```

```
Query OK, 4 rows affected (0.01 sec)
```

```
Records: 4 Duplicates: 0 Warnings: 0
```

```
master> COMMIT;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
master> SHOW BINLOG EVENTS IN 'master-bin.000053' FROM 106\G
```

```
***** 1. row *****
```

```
Log_name: master-bin.000054
```

```
Pos: 106
```

```
Event_type: Query
```

```
Server_id: 1
```

```
End_log_pos: 174
```

```
Info: BEGIN
```

```
***** 2. row *****
```

```
Log_name: master-bin.000054
```

```
Pos: 174
```

```
Event_type: Table_map
```

```
Server_id: 1
```

```
End_log_pos: 215
```

```
Info: table_id: 18 (test.t1)
```

```
***** 3. row *****
```

```

    Log_name: master-bin.000054
      Pos: 215
Event_type: Write_rows
  Server_id: 1
End_log_pos: 264
      Info: table_id: 18 flags: STMT_END_F
***** 4. row *****
    Log_name: master-bin.000054
      Pos: 264
Event_type: Table_map
  Server_id: 1
End_log_pos: 305
      Info: table_id: 18 (test.t1)
***** 5. row *****
    Log_name: master-bin.000054
      Pos: 305
Event_type: Write_rows
  Server_id: 1
End_log_pos: 354
      Info: table_id: 18 flags: STMT_END_F
***** 6. row *****
    Log_name: master-bin.000054
      Pos: 354
Event_type: Xid
  Server_id: 1
End_log_pos: 381
      Info: COMMIT /* xid=23 */
6 rows in set (0.00 sec)

```

这个例子向二进制日志写入两条语句。每个语句以 `Table_map` 事件开始，后面跟着一个 `Write_rows` 事件，该事件保存每个语句变更的 4 行。

可以看到，设置行事件的 `statement-end` 标记能够终止语句。由于语句在某个事务内部，相应的 `Query` 事件也要包含 `BEGIN` 和 `COMMIT` 语句。

行事件的大小通过 `binlog-row-event-max-size` 参数控制，表示它在二进制日志中的最大字节数。这个参数指定的不是行事件的最大大小。如果某行的字节数超过 `binlog-row-event-max-size`，那么 `binlog` 行事件的大小可能比这个参数值大。

## Table\_map 事件

前面提到，`Table_map` 事件将表名映射为标识符，然后用于行事件，但这不是 `Table_`

map 事件的唯一用途。它还包含 master 上的表中字段的基本信息。所以, slave 能够检查 slave 上的表的基本结构, 并与 master 上的结构进行比较, 确保两者匹配, 从而复制继续。

281 Table\_map 事件的基本结构如图 8-12 所示。常用头(所有复制事件都有的头)包含事件的基本信息。然后是提交头提供 Table\_map 事件的特有信息。图 8-12 中的大多数字段都可以从字面上理解, 不过我们进一步看一下其中几个字段。

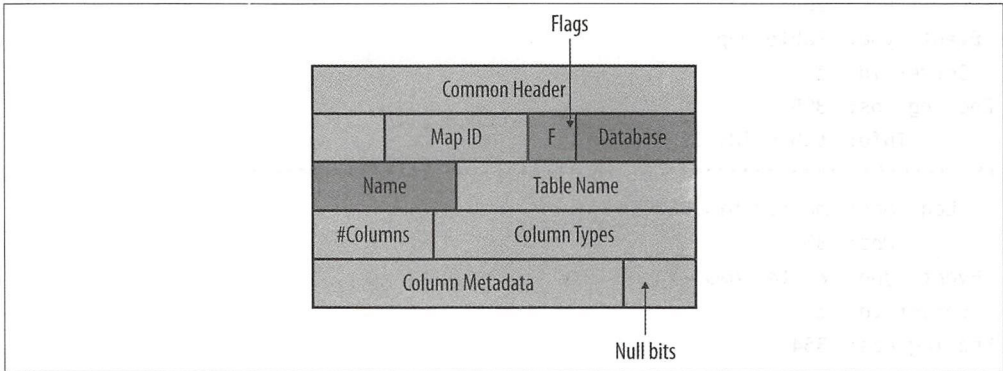


图8-12: Table\_map事件的结构

以下几个字段放在一起表示列类型:

#### 列类型数组

表示所有列的基础类型的数组, 包括整型、字符串类型、小数数据类型, 或其他任何可用类型, 但没有提供列类型的参数。例如, 如果列类型为 CHAR(5), 这个数组包含值 254 (这个常数表示字符串), 但这个字符串的长度 (即 5) 却存储在列元数据中, 后面马上讲到。

#### 空比特数组

表示每个字段是否是 NULL 的比特数组。

#### 列元数据

表示字段元数据的数组, 充实列类型数组的细节信息。每个字段上有哪些元数据取决于字段类型。例如, DECIMAL 类型保存精度和小数, 而 VARCHAR 类型保存字段的最大长度。

综合这 3 个数组, 就得到了字段类型。

不是所有的类型信息都存储在这些数组中。在下面两种情况下, master 和 slave 无法区分两种类型:



- 没有信息说明整型字段是有符号的还是无符号的。这时，slave 在检查表的时候，不知道这个字段是有符号的还是无符号的。
- 没有提供字符串类型的字符集。使用两种不同的字符集进行复制是不支持的，可能导致奇怪的结果，因为不会检查或转换刚刚插入列的字节。

## 行事件的结构

图 8-13 展示了行事件的结构。根据不同的事件类型(写、删除或更新),这个结构稍有不同。

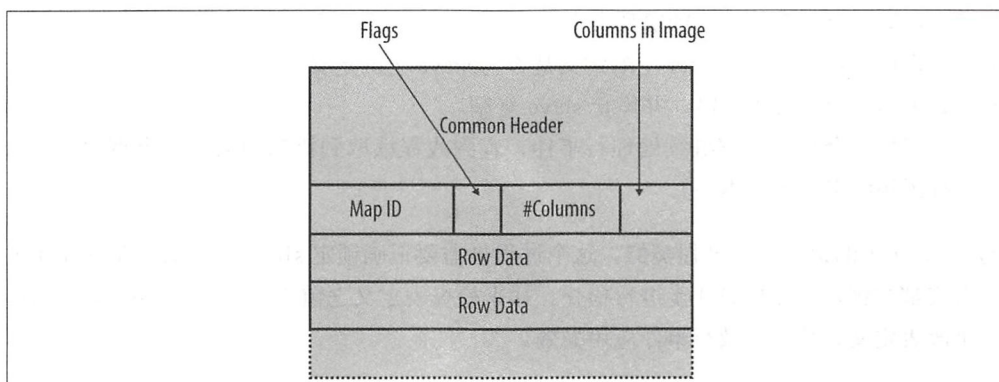


图8-13: 行事件的头

除了表的标识符（即前面表映射事件中的表 ID），该事件还包含以下字段：

### 表宽

master 上表的宽度。同客户端协议一样，这个宽度是基于长度编码的，所以只有一个字节。大多数时候只有一个字节。

### 列位图

表示作为事件的一部分发送的那些列。这个字段使 master 能够对每行选择不同的字段发送。列位图有两种：一种是前映像（before image），一种是后映像（after image）。前映像用于删除和更新操作，而后映像用于写（插入）和更新操作。更多信息参见表 8-3。

表8-3: 行事件及其映像

前映像	后映像	事件
None	Row to insert	Write rows
Row to delete	None	Delete rows
Column values before update	Column values after update	Update rows

## 行事件的执行

因为多个事件可能表示 master 上执行的单个语句，所以 slave 需要保存状态信息，当有并发线程更新同一张表时，保证行事件的正确执行。回想一下，二进制日志中的每个语句都以一个或多个表映射事件开始，跟着一个或多个类型相同的行事件。二进制日志的语句处理步骤如下：

1. 从中继日志中读取各个事件。
2. 如果是表映射事件，SQL 线程将提取表信息，并保存 master 对这个表的定义。
3. 出现第一个行事件时，锁定列表中的所有表。
4. 线程检查每张表在 master 上的定义是否与 slave 上的定义一致。
5. 如果不一致，线程报错，并停止 slave 复制。
6. 按照稍后介绍的步骤继续处理行事件，直到线程读取到语句的最后一个事件（即带有语句结束标志的事件）。

与 master 上的语句执行过程类似，这个过程也需要正确锁定 slave 上的表。第 3 步中的所有表都被锁定，然后第 4 步进行检查。如果检查表定义之前不锁定表，slave 线程可能会更改表定义，从而导致行事件应用失效。

根据事件类型不同，各个行事件包含的行用处不同。对于 Delete\_rows 和 Write\_rows 事件，每行表示一个改变。而 Update\_rows 事件有两行，一行用来正确定位需要更新的行，另一行存储新的行值，所以该事件含有偶数个行，每两行表示一次更新。

拥有前映像（before image）的事件需要经过查找然后正确定位到需要操作的行，比如 Delete\_rows 事件删除行，而 Update\_rows 事件更改行。按照查找优先级递减的顺序，这些查找操作包括：

### 284 主键查询

如果 slave 上的表有主键，就使用主键进行查询操作。这是最快的方法。

#### 索引扫描

如果表中没有定义主键，但定义了索引，就使用索引定位到需要改变的行。扫描索引中的所有行，然后与 master 行进行字段比较。

如果找到了匹配的行，则执行 Delete\_rows 或 Update\_rows 操作。否则，slave 停止复制，报告“找不到正确的行”的错误。

#### 表扫描

如果表上既没有主键也没有索引，则使用全表扫描。

与索引扫描一样，每行都会被扫描，然后与 master 行比较，接着对匹配的行执行删除或更新操作。

使用的是 slave（而不是 master）上的索引或主键来定位正确的行执行删除或更新操作，因此需要注意以下几点：

- 如果 slave 上的表有主键，查询将很快；如果没有，则必须进行全表扫描或索引扫描，相对较慢。
- master 和 slave 上的索引可能不同。

无论采用基于行的复制还是基于语句的复制，最好都使用主键来复制表。

由于基于语句的复制实际上执行了每条语句，所以基于主键的更新和删除也极大地加速了基于语句的复制。

## 事件和触发器

基于语句的复制和基于行的复制，对事件和触发器的执行过程不同。对于事件来说，唯一的区别就是基于行的复制产生行事件而不是查询事件。

而对触发器来说，区别就更大了。

第 4 章中讨论过，基于语句的复制将触发器的定义复制到 slave，所以执行带有触发器的语句时，slave 上也会执行触发器。

而对基于行的复制来说，无论行如何变化——无论变化来自触发器、存储过程、事件或者直接来自于语句本身，由于触发器更新的行已经被复制到 slave，所以触发器不需要在 slave 上再次执行。实际上，如果在 slave 上执行触发器会导致不正确的结果。

285

示例 8-26 定义了一个带有触发器的表。

示例8-26：表及触发器的定义

```
CREATE TABLE log (  
    number INT AUTO_INCREMENT PRIMARY KEY,  
    user CHAR(64),  
    brief TEXT  
);
```

```
CREATE TABLE user (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    email CHAR(64),
```

```
password CHAR(64)
);
```

```
CREATE TRIGGER tr_update_user AFTER UPDATE ON user FOR EACH ROW
INSERT INTO log SET
    user = NEW.email,
    brief = CONCAT("Changed password from '",
        OLD.password, "' to '",
        NEW.password, "'");
```

```
CREATE TRIGGER tr_insert_user AFTER INSERT ON user FOR EACH ROW
INSERT INTO log SET
    user = NEW.email,
    brief = CONCAT("User '", NEW.email, "' added");
```

定义好表和触发器后，顺序执行下面的语句。

```
master> INSERT INTO user(email,password) VALUES ('mats@example.com', 'xyzy');
Query OK, 1 row affected (0.05 sec)
```

```
master> UPDATE user SET password = 'secret' WHERE email = 'mats@example.com';
Query OK, 1 row affected (0.01 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
master> SELECT * FROM log;
```

number	user	brief
1	mats@sun.com	User 'mats@example.com' added
2	mats@sun.com	Changed password from 'xyzy' to 'secret'

286 2 rows in set (0.00 sec)

当然，这么做不太安全，但至少可以说明问题。那么，使用基于行的复制时，二进制日志将如何反映这些更新呢？

```
master> SHOW BINLOG EVENTS IN 'mysqld1-bin.000054' FROM 2180;
```

Log_name	Pos	Event_type	Server_id	End_log_pos	Info
master-bin...54	2180	Query	1	2248	BEGIN
master-bin...54	2248	Table_map	1	2297	table_id: 24 (test.user)
master-bin...54	2297	Table_map	1	2344	table_id: 26 (test.log)



```

|master-bin...54|2344|Write_rows |      1|      2397|table_id: 24      |
|master-bin...54|2397|Write_rows |      1|      2471|table_id: 26 flags:    |
|              |      |      |      |      |      STMT_END_F      |
|master-bin...54|2471|Query      |      1|      2540|COMMIT                |
|master-bin...54|2540|Query      |      1|      2608|BEGIN                 |
|master-bin...54|2608|Table_map  |      1|      2657|table_id: 24 (test.user)|
|master-bin...54|2657|Table_map  |      1|      2704|table_id: 26 (test.log)|
|master-bin...54|2704|Update_rows|      1|      2783|table_id: 24          |
|master-bin...54|2783|Write_rows |      1|      2873|table_id: 26 flags:    |
|              |      |      |      |      |      STMT_END_F      |
|master-bin...54|2873|Query      |      1|      2942|COMMIT                |
+-----+-----+-----+-----+-----+-----+
12 rows in set (0.00 sec)

```

可见，每个语句都是一个仅包含单个语句的单独事务。语句更改两张表（即 `test.user` 表和 `test.log` 表），因此在二进制日志中，语句的开始处有两个表映射事件。事件复制到 `slave` 后直接执行，不考虑触发器，从而避免了 `slave` 表再次执行触发器。

## 基于行的复制中的过滤

基于语句的复制和基于行的复制对过滤的处理也不相同。回忆一下第 4 章，基于语句的复制是在整个语句上完成过滤的（即要么所有语句都执行，要么所有语句都不执行），因为只执行部分语句是不可能的。对数据库过滤来说，使用当前数据库而不是正在操作的表所在的数据库。

而基于行的复制提供了更多选择。因为特定表的每一行都会被复制，所以可以过滤正在更新的真实表，甚至可以基于任意条件过滤行。因此，基于行的复制的过滤是基于真正发生变化的表，而不是语句的当前数据库。

考虑一下，如果将 `slave` 上的配置改为忽略 `ignore_me` 数据库，那么如何过滤呢？对于 287 基于语句的复制和基于行的复制，执行下面的语句分别会产生什么结果？

```
USE test; INSERT INTO ignore_me.t1 VALUES (1),(2);
```

基于语句的复制执行该语句，而基于行的复制则忽略表 `t1` 上的改变，因为 `ignore_me` 数据库已被忽略。

接着执行下面的多表更新语句又将怎样呢？

```
USE test; UPDATE ignore_me.t1, test.t2 SET t1.a = 3, t2.a = 4 WHERE t1.a = t2.a;
```

基于语句的复制执行该语句，以为 `ignore_me.t1` 表存在（实际上该表并不存在，因为数据库已被忽略），并更新 `ignore_me.t1` 表和 `test.t2` 表。而基于行的复制只更新 `test.t2` 表。

## 语句的部分执行

注意，如果不考虑失效、崩溃及非确定性行为，通常基于语句的复制性能很好。最坏的情况是发生失效或崩溃，这时几乎总是导致语句部分执行。

如果 `UPDATE`、`DELETE` 或 `INSERT` 语句影响的行数被人为地限定，也会导致语句部分执行。例如，使用 `LIMIT` 从句，或者非事务型表上的执行中断，比如，重复键错误导致执行中止，这时语句将部分执行。

这些情况下，语句所描述的变更只是部分应用到某个初始的行集。由于 `master` 和 `slave` 上这些行的顺序不同，所以语句应用的行集也不同。

`MyISAM` 按照行的插入顺序维护所有行的顺序，如果发生部分执行，可以帮助你更新相同的行集。但不幸的是，事实并不是这样。如果 `slave` 通过逻辑备份或从备份中恢复的方式从 `master` 克隆数据，行的插入顺序可能会发生改变。

通常使用 `ORDER BY` 从句可以解决这个问题，但这也不完全安全，因为还是有可能由于崩溃而导致语句部分执行。

## 288 部分行复制

前面讲过，`Write_rows`、`Delete_rows` 和 `Update_rows` 事件各自包含一个列位图，说明事件包含的行中有哪些列。注意，前映像和后映像各自有一个位图。

在 `MySQL 5.6.2` 之前，只有 `MySQL` 集群引擎限制写入日志的列，但是从 `5.6.2` 开始，可以通过 `binlog-row-image` 参数控制哪些列写入日志。这个参数有三个值：`full`、`noblob` 和 `minimal`。

### full

这是 `binlog-row-image` 的默认值，即复制全部列。在 `5.6.2` 之前，行总是按照这种方式写入日志的。

### noblob

忽略 `blob` 类型，除非它们需要被更新。

minimal

只有主键（前映像中的）和更改值的那些列（后映像中的）被写入二进制日志。

采用 full 作为默认值，是因为 master 和 slave 上的索引可能不同，而且可能需要 master 上的非主键列才能正确定位 slave 上的行。

观察示例 8-27，分别在 master 和 slave 上定义了不同的表，但是唯一的不同就是索引不同。master 以 id 列为主键是为了使用自增量，而 slave 使用 email 列作为主键。

在这个例子中，将 binlog-row-image 设置为 minimal，可以将所有的 id 列写入二进制日志，但这个列不能用来定位 slave 上的正确行，这样复制就会失败。而我们期望即使出错复制也能运行，所以将 binlog-row-image 的默认值设置为 full。

如果在 master 和 slave 上使用完全一样的索引（或者至少 slave 上的索引列在 master 上也被索引），就可以将 binlog-row-image 设置为 minimal，而且还减少了二进制日志的大小，节省了空间。

那么，noblob 是什么意思呢？这是一个中间值。即使 master 和 slave 上有不同的索引，blob 也很少是索引的一部分。由于 blob 常常很占空间，假设 blob 不会被索引的前提下，使用 noblob 跟 full 差不多安全。

示例 8-27: master 和 slave 上有不同索引的表

289

```
/* master 上的表定义 */
CREATE TABLE user (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email CHAR(64),
    password CHAR(64)
);

/* slave 上的表定义 */
CREATE TABLE user (
    id INT,
    email CHAR(64) PRIMARY KEY,
    password CHAR(64)
);
```

## 小结

本章结束了关于 MySQL 复制的若干章节，讨论了高级复制的相关内容，比如如何更加

健壮地将 slave 提升为 master，崩溃后如何避免数据库损坏的技巧和技术，解释了多源复制的配置及其需要考虑的问题，最后详细介绍了基于行的复制。

在下一章中，我们将学习构建健壮的数据中心的另一类问题，包括监控、存储引擎的性能调优和复制。

Joel 吃完午饭回来，在大厅的走廊上碰见了老板，“您好，Summerson 先生。”

“你好啊，Joel。”

“您看过我的报告了吗？”

“看过了，Joel。做得不错嘛。我已经分发给其他几个部门，看看他们有什么意见。我想把它加到我们的 SOP 手册中。”

Joel 心想 SOP 的意思是标准的操作流程。

“我已经叫他们看过以后把意见发给你。要想加到 SOP 中可能还需要一些语言上的润饰，我想你肯定行。”

“谢谢，先生。”

Summerson 先生点点头，拍了拍 Joel 的肩膀，然后走了。



# MySQL 集群

一阵平缓的敲门声提醒 Joel 有人来了，他一抬头看见满脸愁容的 Summerson 先生。

“这次要靠你了，我们遇到麻烦了。”

Joel 什么也没说，他在想“靠我”是什么意思。目前 Summerson 先生已经给他安排了很多紧张的工作。

“我们刚刚得知，有一个新客户想在实时和‘99999’的环境中使用我们最新的数据库应用程序。”

“始终运行、不停机？”

“是的。我知道 MySQL 很可靠，但我们现在没时间把应用程序的后台改为容错的数据库服务器。”

Joel 记得书里面有一章介绍了 MySQL 的特殊技术，不知道能不能派上用场。他决定试一试：“我们可以采用集群技术。”

“集群？”

“是的，MySQL 有一个集群版本，它是一个容错的数据库系统。我记得它可以在某些相当苛刻的环境下工作，像远程通信等……”

Summerson 先生眼前一亮，仿佛站得更直了，接着他做了一个决定性的发言。“太好了，明天早上给我做个报告。我需要知道成本和硬件方面的需求和限制，不要有所保留。如果这么做行得通的话那就去做，但我不想凭感觉冒险。”

“我马上就去做。”Joel 不知道自己这次摊上什么事儿了。Summerson 先生离开后，他叹了口气，然后打开了心爱的 MySQL 书。“这可能是我目前遇到的最大的挑战了。”他说。

高性能、高可用性、冗余和可扩展性都是数据库规划时需要考虑的重要因素，常常采用商用高可用性硬件和均衡负载等方案来寻求改善复制拓扑的方法。尽管这样常常可以满足大部分需求，但是如果需要无单点故障的方案，而且要求极高的吞吐量，上线时间达到 99.999%，那么可以考虑使用 MySQL 集群技术。

本章将介绍 MySQL 集群技术的相关概念，演示如何启动和停止一个简单的集群，讨论 MySQL 集群使用过程中的关键问题，包括高可用性、分布式数据和数据复制。首先将解释什么是 MySQL 集群，以及它与普通的 MySQL 服务器有何不同。

## 什么是 MySQL 集群

MySQL 集群是一个无共享的（shared-nothing）、分布式节点架构的存储方案，其目的是提供容错性和高性能。数据在单个数据节点上存储和复制，每个数据节点运行在独立的服务器上并维护数据的一份副本。每个集群还有管理节点。数据更新使用读已提交隔离级别（read-committed）来保证所有节点数据的一致性，使用两阶段提交机制（two-phased commit）保证所有节点都有相同的数据（如果任何一个写操作失败，则更新失败）。

MySQL 集群的最初实现是将所有信息都保存在主存内，没有任何永久性存储。后来 MySQL 集群允许数据存储在磁盘上。通过存储引擎层将 MySQL 服务器作为查询引擎，可以使 MySQL 集群的性能达到最佳。这样就可以将 MySQL 应用透明地迁移到 MySQL 集群中去。

无共享的对等节点使得某台服务器上的更新操作在其他服务器上立即可见。传播更新使用一种复杂的通信机制，这一机制专用来提供跨网络的高吞吐量。该架构通过多个 MySQL 服务器分摊负载，从而最大程度地达到高性能，通过在不同位置存储数据保证高可用性和冗余。

## 术语和组件

MySQL 集群的典型部署是在某个网络的不同机器上安装集群组件。因此，MySQL 集群又称网络数据库（network database，NDB）。这里“MySQL 集群”指的是 MySQL 服务器和 NDB 组件，而“NDB”或“NDB 集群”则特指集群组件。

MySQL 集群是一个数据库系统，使用 MySQL 服务器作为前端来支持标准的 SQL 查询。  
名为 NDBCluster 的存储引擎是连接 MySQL 服务器和集群技术的接口，这个关系经常容易混淆。如果没有 NDBCluster 组件，就不能使用 NDBCluster 存储引擎。但是，没有 MySQL 服务器也可以使用 NDB 集群技术，只不过需要一些 NDB API 的底层编程。

NDB API 是面向对象的，实现了索引、扫描、事务和事件处理。你可以编写检索、存储和管理集群数据的应用。NDB API 还提供了面向对象的错误处理机制，允许有序的关机和故障恢复操作。如果你是一个开发者，想了解更多关于 NDB API 的信息，请参见 MySQL NDB API 的在线文档 (<http://bit.ly/ndb-api>)。

## MySQL 集群和 MySQL 有何不同

你可能会问：“集群和复制之间有什么区别呢？”集群的定义很多，通常认为集群包含成员、消息、冗余和自动故障转移等功能，而复制仅仅是一个服务器向另一个服务器发送消息（数据）的方式。我们先讨论集群内部的复制（又称本地复制），后面再详细讲述 MySQL 复制。

## 典型配置

MySQL 集群有如下三层。

- 应用程序层：负责与 MySQL 服务器通信的各种应用程序。
- MySQL 服务器层：处理 SQL 命令，并与 NDB 存储引擎通信的 MySQL 服务器。
- NDB 集群组件层：NDB 集群组件（即数据节点），负责处理查询，然后将结果返回给 MySQL 服务器。



每一层都可以独立地纵向扩展(scale up),即通过更多的服务器进程来提高性能。

图 9-1 显示了典型的集群配置概念图。

应用程序连接到 MySQL 服务器,通过存储引擎层(如 NDB 存储引擎)访问 NDB 集群组件。接下来将详细讨论 NDB 集群组件。

配置的种类很多。可以使用多个 MySQL 服务器来连接单个 NDB 集群，甚至可以通过 MySQL 复制连接到多个 NDB 集群。后面再讨论这些配置。

294

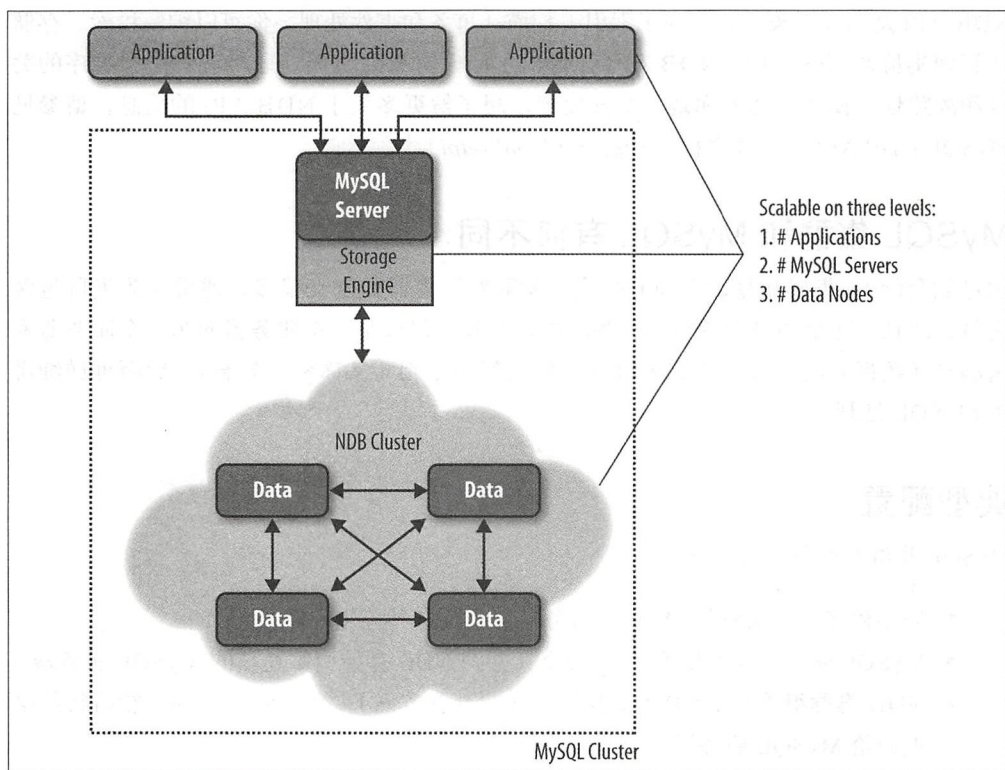


图9-1: MySQL集群

## MySQL 集群的特点

为了实现最高性能、高可用性和冗余等目标，数据在集群内部的对等数据节点之间相互复制。数据复制采用同步机制，数据存储在多个数据节点上，每个数据节点连接到所有其他数据节点上。

295



集群之间也可以复制数据，这时需要使用 MySQL 复制技术，它是异步的，而不是同步的。前面的章节讲过，异步复制意味着更新 slave 的时候有一定延迟，slave 不会将更新的进度报告给 master，所以你不能像在单个 MySQL 集群中那样期望复制架构中的所有服务器之间完全一致。

MySQL 集群有一些创建高可用性系统的专用功能，主要包括：

### 节点恢复

通过通信丢失或心跳失效来检测数据节点故障，还可以将节点配置为从其他节点的



数据副本中自动重启。故障和恢复可以包含单个或多个存储节点。节点恢复又称本地恢复。

#### 日志

通常数据更新时，向每个数据节点的日志中写入一份数据变化事件的副本。该日志可以将数据恢复到某个时间点。

#### 检查点

集群支持两种检查点，即本地检查点和全局检查点。本地检查点删除日志文件的尾部。当所有数据节点上的日志都被刷新到磁盘时将创建全局检查点，在磁盘上创建一个所有节点数据的事务一致性快照。通过这种方式，检查点允许整个系统从某个已知的同步点恢复所有节点。

#### 系统恢复

如果整个系统非正常关闭，可以使用检查点和变更日志进行恢复。通常从某个已知的同步点将数据从磁盘复制到内存。

#### 热备份及恢复

可以在不干扰事务执行的情况下创建每个数据节点的同步备份，包括数据库中对象的元数据、数据本身和当前事务日志。

#### 无单点故障

此架构保证了任何一个节点失效都不会导致数据库系统崩溃。

#### 故障转移

为了保证节点可恢复，所有事务的提交都采用读已提交隔离级别和两阶段提交机制。这样事务就是双重安全的，也就是说，事务到达用户之前存储在两个不同的地方。

◀ 296

#### 分区

数据在数据节点之间被自动分区。从 MySQL 5.1 版本开始，集群支持用户自定义分区。

#### 联机操作

可以无中断地联机执行很多维护操作。通常需要正常关闭服务器或者给数据加锁。例如，联机添加新的数据节点，改变表结构，甚至重组集群中的数据。

关于 MySQL 集群的更多信息，可参见 MySQL 集群文档 (<http://bit.ly/sql-cluster>)，其中包含不同集群版本的参考手册。

## 本地和全局冗余

使用两阶段提交协议可以创建局部冗余（在某个集群内部）。原则上，如果每个节点都同意变更，则提交事务。在同意阶段，每个节点都保证下一轮有足够的资源来提交变更。在 NDB 集群中，MySQL 服务器的提交协议允许多节点变更。NDB 集群也有一个两阶段提交的优化版本，减少了同步复制过程中发送的消息量。这种两阶段协议保证数据冗余地存储在多个数据节点上，这种状态称为本地冗余（local redundancy）。

全局冗余（global redundancy）在集群之间使用 MySQL 复制，这会在整个复制拓扑中建立两个节点。前面讲过，MySQL 复制是异步的，因为在复制事件到来或执行的时候没有确认。图 9-2 解释了本地冗余和全局冗余的区别。

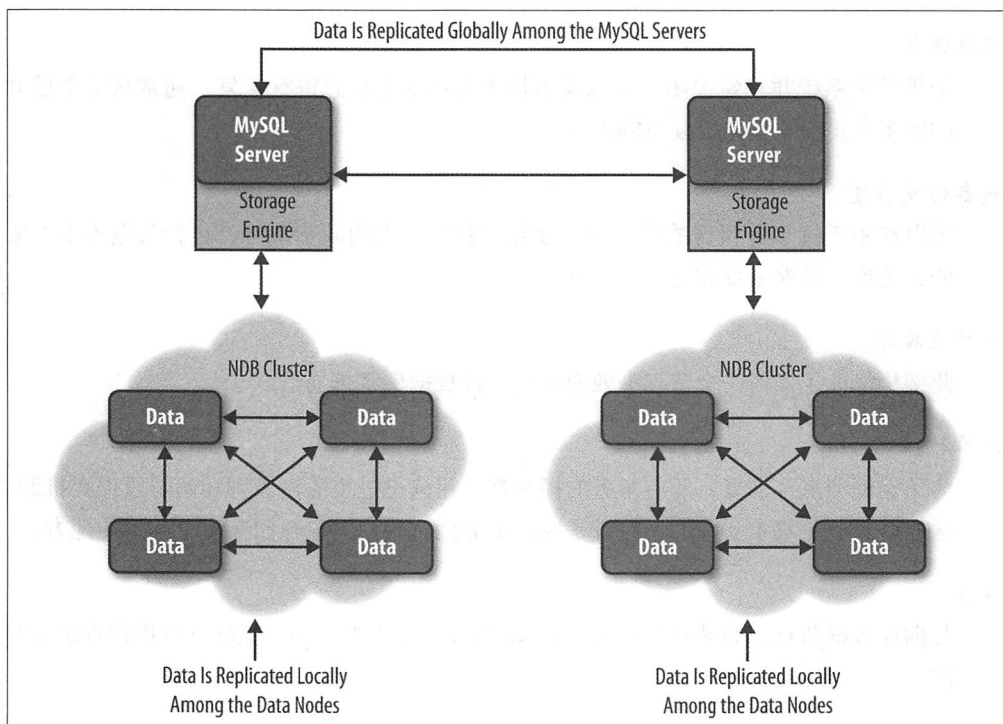


图9-2：本地冗余和全局冗余

297

## 日志处理

MySQL 集群实现了两种类型的检查点：（1）本地检查点，用于清除部分重做日志；（2）全局检查点，主要用于不同数据节点之间的同步。全局检查点对于复制来说很重要，因为它形成了事务组之间的边界，称为 epoch。每个 epoch 是集群之间复制的单位。实际上，

MySQL 复制把两个连续的全局检查点之间的事务组看成单个事务。

## 冗余和分布式数据

数据冗余用副本（replica）实现，每个副本包含数据的一份拷贝。这样集群就可以容错：如果任何一个节点失效，仍然可以访问数据。当然，集群中的副本越多，其容错性就越好。

可以指定集群中的数据副本数目（NoOfReplicas）。需要多少副本，就设置相应数量的数据节点。还可以利用分区将数据分布到各个数据节点，每个数据节点仅存储部分数据，这样查询更快。由于数据有多份副本，即使节点故障仍可以查询数据，也可以恢复丢失的节点（因为其他副本中数据还存在）。为此，每一个副本都需要多个数据节点来存储。例如，如果有 2 个副本，且已分区，那么你至少需要 4 个数据节点（每个副本需要 2 个数据节点）。

### 裂脑综合征

298

如果一个或多个节点失效，可能其他数据节点之间也不能通信，这样两组数据节点就处于“裂脑”状态。这类情况很麻烦，因为从理论上说，每组数据节点构成了一个独立集群。

为此，需要一个网络分区算法解决各组数据节点之间的竞争，每组独立进行选举。节点数目较少的组将重启，然后分别将该组中的每个节点添加到节点数目较多的组。

如果两组节点数目相当，仍然不能解决问题。例如将 4 个节点分成两组，每组 2 个节点，怎么选择组呢？这时可以定义一个仲裁器，规定第一个成功连接到仲裁器的组获胜。

仲裁器可以是 MySQL 服务器（SQL 节点）或管理节点。为了达到高可用性，最好将仲裁器放在非数据节点的系统上。

带有仲裁器的网络分区算法在 MySQL 集群中是完全自动化的，“少数”的定义与节点组相关，与仅对节点进行计数的方法相比，它的可用性更高。

## MySQL 集群的架构

MySQL 集群由一个或多个 MySQL 服务器组成，这些 MySQL 服务器通过 NDB 存储引擎与 NDB 集群通信。NDB 集群由以下组件构成：存储和检索数据的数据或存储节点，

以及一个或多个管理节点，负责协调数据节点的启动、关闭和恢复。大部分 NDB 组件都作为守护进程执行，MySQL 集群还提供操作守护进程的客户端实用程序。下面列出了一些守护进程和实用程序。图 9-3 描绘了这些组件相互之间是如何通信的。

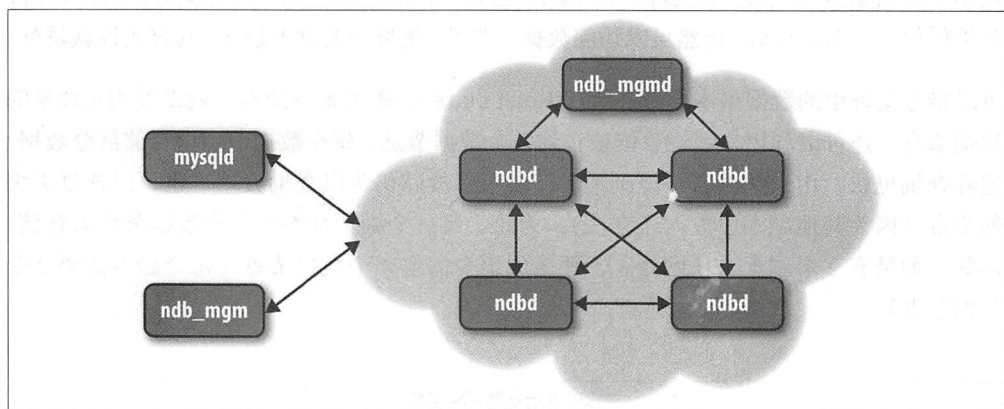


图9-3: MySQL集群组件

299

`mysqld`

MySQL 服务器。

`ndbd`

数据节点。

`ndbmtid`

多线程数据节点。

`ndb_mgmd`

集群的管理服务器。

`ndb_mgm`

集群的管理客户端。

每个名为 `mysqld` 的 MySQL 服务器通常都支持一个或多个 SQL 查询应用，然后接收来自数据节点的返回结果。讨论 MySQL 集群的时候，有时 MySQL 服务器又称 SQL 节点。

数据节点是一系列 NDB 守护进程，负责存储和检索内存或磁盘上（由配置决定）的数据。数据节点安装在集群中的各个服务器上。还有一个名为 `ndbmtid` 的多线程数据节点守护进程，运行在支持多核 CPU 的平台上。如果在现代的多核 CPU 专用服务器上使用多线程数据节点，可以提高数据节点的性能。



管理守护进程 `ndb_mgmd` 运行在服务器上，负责读入配置文件，然后将信息分发到集群中的所有节点上。NDB 管理客户端实用程序 `ndb_mgm` 可以检查集群的状态，开始备份，然后执行其他管理功能。这个客户端运行在一个方便管理的主机上，并与守护进程通信。

很多实用程序可以简化维护工作，下面给出了几个常用的实用程序。要想得到完整列表，请查阅 NDB 集群文档 ([http://bit.ly/cluster\\_ndb](http://bit.ly/cluster_ndb))。

`ndb_config`

抽取已有节点的配置信息。

`ndb_delete_all`

删除 NDB 表中的所有行。

`ndb_desc`

描述 NDB 表（就像 `SHOW CREATE TABLE` 命令一样）。

`ndb_drop_index`

删除 NDB 表的索引。

`ndb_drop_table`

删除 NDB 表。

`ndb_error_reporter`

诊断集群中的错误和问题。

`ndb_redo_log_reader`

检查并输出集群的重做日志。

`ndb_restore`

执行集群的恢复，使用 NDB 管理客户端进行备份。

## 如何存储数据

MySQL 集群将所有的索引列都保存在主存中，其他非索引列可以存储在内存中，或者存储到带有内存页面缓存的磁盘上。磁盘可以比内存存储更多的非索引列数据。

如果数据发生改变（通过 `INSERT`、`UPDATE`、`DELETE` 等），MySQL 集群将发生改变的记录写入重做日志，然后通过检查点定期将数据写入磁盘。前面讲过，通过日志和检查点可以从磁盘进行故障恢复。但是，由于重做日志是异步提交的，所以故障期间可能有少量事务丢失。为了减少事务丢失，MySQL 集群实现了延迟写入（默认延迟两秒，可配置），这样就可以在故障发生时完成检查点写入，而不会丢失最后一个检查点。一般单个数据

节点故障不会导致任何数据丢失，因为集群内部采用同步数据复制。

301 MySQL 集群的表在内存中维护，只有在向重做日志中写入数据和执行必要的检查点操作时，集群才需要访问磁盘存储。因为写日志和检查点都是顺序操作，很少出现随机访问模式，所以比起关系数据库中传统的磁盘缓存机制，MySQL 集群可以通过有限的磁盘硬件得到更高的写吞吐率。

使用下面的公式可以计算出一个数据节点需要的内存大小。数据库的大小是行的大小乘以每个表的行数。记住，如果使用磁盘存储非索引字段，在计算内存需求时只需计算索引字段。

$$(\text{数据库的大小} * \text{副本的数量} * 1.1) / \text{数据节点的数量}$$

这是用于粗略计算的一个简单公式。在规划集群的内存需求时，可以查阅在线 MySQL 集群参考手册。

还可以在大部分分发部署中使用 Perl 脚本 *ndb\_size.pl*，连接到一个正在运行的 MySQL 服务器，遍历数据库中已有的表，然后计算 MySQL 集群可能需要的内存。这个脚本很方便，因为它可以让你先在一个普通 MySQL 服务器上创建和填充表，然后检查内存配置，配置系统，最后将数据装载到集群中去。这同样有利于定期运行，从而避免数据库模式改变带来的内存问题，还能让你了解内存的使用情况。示例 9-1 描述了含有一张表的简单数据库的报表示例。为了得出数据库的总大小，我们用汇总数据中的数据行大小乘以行数。在这个例子中，数据和索引每行大小为 84 字节 (MySQL 版本为 5.1)。如果有 64 000 行，则需要 5 376 000 字节的内存来存储这张表。



如果脚本报错没有 *Class/MethodMaker.pm* 模块，则需要在自己的系统上安装这个类。例如，在 Ubuntu 上使用如下命令安装：

```
sudo apt-get install libclass-methodmaker-perl
```

示例9-1：使用 *ndb\_size.pl* 检查数据库的大小

```
$ ./ndb_size.pl \  
> --database=cluster_test --user=root  
ndb_size.pl report for database: 'cluster_test' (1 tables)  
-----  
Connected to: DBI:mysql:host=localhost
```

302 Including information for versions: 4.1, 5.0, 5.1

```
cluster_test.City  
-----
```

DataMemory for Columns (\* means var sized DataMemory):

Column Name	Type	Varsized	Key	4.1	5.0	5.1
district	char(20)			20	20	20
population	int(11)			4	4	4
ccode	char(3)			4	4	4
name	char(35)			36	36	36
id	int(11)		PRI	4	4	4
				--	--	--
Fixed Size Columns DM/Row				68	68	68
Varsize Columns DM/Row				0	0	0

DataMemory for Indexes:

Index Name	Type	4.1	5.0	5.1
PRIMARY	BTREE	N/A	N/A	N/A
		--	--	--
Total Index DM/Row		0	0	0

IndexMemory for Indexes:

Index Name	4.1	5.0	5.1
PRIMARY	29	16	16
	--	--	--
Indexes IM/Row		29	16

Summary (for THIS table):

	4.1	5.0	5.1
Fixed Overhead DM/Row	12	12	16
NULL Bytes/Row	0	0	0
DataMemory/Row	80	80	84

(Includes overhead, bitmap and indexes)

Varsize Overhead DM/Row	0	0	8
Varsize NULL Bytes/Row	0	0	0
Avg Varside DM/Row	0	0	0

No. Rows	3	3	3
----------	---	---	---

Rows/32kb DM Page	408	408	388
Fixedsize DataMemory (KB)	32	32	32

Rows/32kb Varsize DM Page	0	0	0
Varsize DataMemory (KB)	0	0	0

Rows/8kb IM Page	282	512	512
IndexMemory (KB)	8	8	8

303 \* indicates greater than default

Parameter	Default	4.1	5.0	5.1
DataMemory (KB)	81920	32	32	32
NoOfOrderedIndexes	128	1	1	1
NoOfTables	128	1	1	1
IndexMemory (KB)	18432	8	8	8
NoOfUniqueHashIndexes	64	0	0	0
NoOfAttributes	1000	5	5	5
NoOfTriggers	768	5	5	5

示例 9-1 以一个很简单的表为例，不仅输出了行的大小，还有数据库中表的统计数据。这个报告还给出了索引统计数据，索引是集群高性能的关键机制。

这个脚本也展示了 MySQL 不同版本之间不同的内存需求。如果你用的是老版本的 MySQL 集群，就会看到其中的区别。

## 分区

数据分区是 MySQL 集群的一个重要方面。MySQL 集群将数据水平分区，即使用某种函数将行自动分布到各个数据节点，通常基于表关键字上的哈希算法实现。在早期的 MySQL 版本中，软件使用一种内部分区机制，而 MySQL 5.1 及其后的版本允许提供自定义函数进行数据分区。如果你使用自己的分区函数，就要创建一个函数来保证数据平均分布到各个数据节点之间。



如果表没有主键，MySQL 集群就会添加一个代理主键。

分区可以使 MySQL 集群获得更好的查询性能，因为分区支持数据节点之间的分布式查询。因此，在几个节点之间收集数据时，查询的返回结果会比单个节点更快。例如，在每个数据节点上执行下面的查询，对每个节点上的列求和，然后汇总：

```
SELECT SUM(population) FROM cluster_db.city;
```

数据分布在数据节点之间，如果数据有多个副本，就可以防止故障的发生。如果你想使用分区将数据分布到多个数据节点上以实现并行查询，还要保证每行至少有两份副本，



这样集群才是容错的。

## 事务管理

MySQL 集群不同于 MySQL 服务器的另一个方面就是关于事务型数据的操作。前面提过, MySQL 集群协调数据节点之间的事务型变更, 通过两个子过程实现, 即事务协调器 (transaction coordinator) 和本地查询处理器 (local query handler)。

事务协调器处理全局的分布式事务和其他数据操作。本地查询处理器管理集群数据节点本地的数据和事务, 并协调处理数据节点的两阶段提交。

每个数据节点都可以是事务协调器 (这是可配置的)。当应用程序执行一个事务时, 集群就会连接到其中一个数据节点的事务协调器, 默认选择最近的数据节点。如果有多个同样距离的可用连接, 就采用轮转法 (round-robin) 选择事务协调器。

接下来, 被选中的事务协调器向每个数据节点发送查询, 然后各个节点的本地查询处理器执行这个查询, 并与事务协调器一起协调两阶段提交。一旦所有数据节点都验证了这个事务, 事务协调器就会验证 (提交) 这个事务。

MySQL 集群支持读已提交事务隔离级别, 即如果事务执行期间发生改变, 只有已提交的变更在事务过程中是可读的。这样, MySQL 集群保证了事务运行过程中的数据一致性。

要了解更多关于事务在 MySQL 集群中是如何工作的, 以及事务的重要局限性, 参见在线 MySQL 参考手册中的 MySQL 集群这一章。

## 联机操作

在 MySQL 5.1 及以后的版本中, 可以联机 (在线) 执行某些操作, 也就是说不需要关闭服务器, 也不用将系统或数据库部分加锁。下面简单列出了 MySQL 集群可用的一些联机操作, 并给出了每个功能的版本信息。

### 备份 (5.0 及之后的版本)

可以使用 NDB 管理控制台来执行快照备份 (非阻塞操作), 从而创建集群数据的一个备份。这个操作包括复制元数据 (即所有表的名字和定义)、表数据和事务日志 (即变更的历史记录)。这与 *mysql-dump* 备份不同, 因为它不使用表扫描来读取记录。可以使用特有的 *ndb\_restore* 实用程序来恢复数据。

◀ 305

### 添加和删除索引 (5.1 及之后的版本)

可以使用 ONLINE 关键字联机执行 CREATE INDEX 或 DROP INDEX 命令。当请求联机操作时, 这个操作是非复制的, 即不会产生数据的副本和索引, 这样以后就不需要重

建索引。这样做的好处是在执行 alter table 操作的时候，正在被修改的表不会被加锁导致其他 SQL 节点不可读，从而事务可以继续进行。但是，对于执行 alter 操作的 SQL 节点上的其他查询来说，这个表仍然是锁定的。



在 MySQL 5.1.7 及以后的版本中，只有当索引列的宽度可变时，添加和删除索引操作才是联机执行的。

#### 修改表（6.2 及以后的版本）

使用 ONLINE 关键字可以联机执行 ALTER TABLE 语句。这同样也是非复制的，和联机添加索引的好处一样。此外，在 MySQL 7.0 及以后的版本中，可以使用 REORGANIZE PARTITION 命令跨分区联机重组数据，但是不能加 INTO(partition\_definitions)选项。



目前还不支持改变默认字段值或数据类型的联机操作。

#### 添加数据节点和节点组（7.0 及以后的版本）

可以联机管理数据节点扩展，包括横向扩展或故障之后的节点替换。这个过程在参考手册中有更加详细的描述。简单地说，包括修改配置文件，执行 NDB 管理守护进程的轮流重启，执行已有数据节点的轮流重启，启动新的数据节点，然后进行分区重组。

关于 MySQL 集群的更多信息及其架构和特性，参见 MySQL 集群的白皮书（<http://bit.ly/mysql-wps>），里面还有很多其他 MySQL 的相关话题。

306

## 配置实例

本节中将展示一个 MySQL 集群的配置实例，包括分别运行在两个系统上的两个数据节点，以及运行在第三个系统上的 MySQL 服务器和 NDB 管理节点。这个例子简化了数据节点的设置，如图 9-4 所示。

如图 9-4 所示，一个节点上既包含 NDB 管理守护进程，同时还是 SQL 节点（即 MySQL 服务器），还有两个数据节点，各自运行在自己的系统中。你需要至少三台计算机来完成这样一个基础的 MySQL 集群配置，以提高可用性或性能。

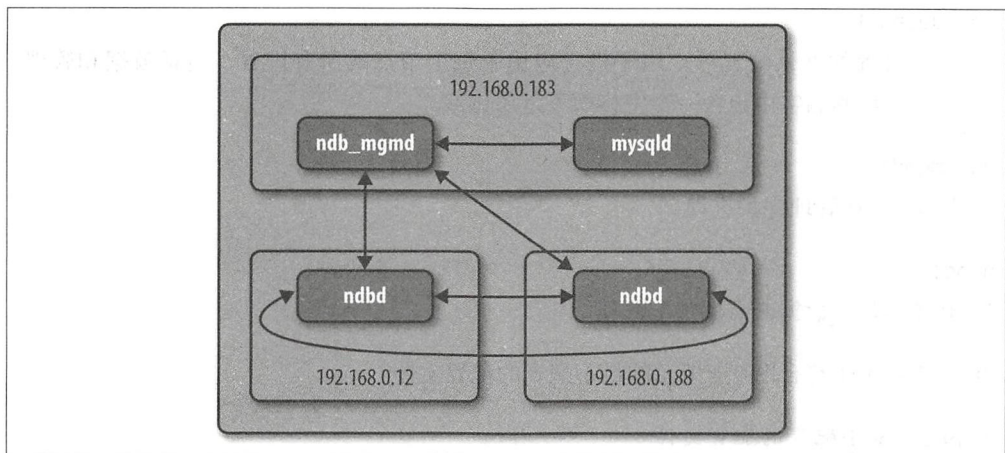


图9-4: 集群配置的简单实例

这是 MySQL 集群的最小配置。如果将副本的数量设为 2，则这个最小配置可以容错。如果将副本的数量设为 1，为获得更好的性能，这个配置可以支持分区，但不能容错。

一般来说，同一个节点既运行 NDB 管理守护进程又作为 MySQL 服务器是允许的，但是如果数据节点的数目很多，或者你想保证最大容错，可能需要将这个守护进程迁移到另一个系统上去。

## 入门

从 MySQL 下载页面 (<http://bit.ly/dl-mysql>) 上可以下载 MySQL 集群，跟 MySQL 服务器一样，它也是开源的。可以下载二进制版本或者常见平台的安装文件，还可以下载源码，307 然后在自己的平台上构建集群。一定要检查主机操作系统需要注意的具体问题。

按照在线 MySQL 参考手册上的正常安装过程进行，除了特殊目录以外，NDB 工具与 MySQL 服务器的二进制文件安装在同一位置。

在介绍实例之前，让我们先重温一下关于配置 MySQL 集群的常用概念。集群配置由 NDB 管理守护进程维护，从配置文件中读取（初始化）配置。可以使用多个参数调整集群的各个部分，但目前我们只考虑最小配置。

配置文件由多个部分组成，至少包含下面几项：

[mysqld]

MySQL 服务器和 SQL 节点的配置文件的常见配置项。

[ndb default]

全局设置的默认配置项，用于指定应用于每个节点的所有设置，包括数据和管理。  
注意此项的名字中包含一个空格而不是下画线。

[ndb\_mgmd]

用于 NDB 管理守护进程。

[ndbd]

必须为每个数据节点都加上该项。

示例 9-2 给出了符合图 9-4 所示的最小配置的配置文件。

示例9-2: 最小配置的配置文件

```
[ndbd default]
```

```
NoOfReplicas= 2
```

```
DataDir= /var/lib/mysql-cluster
```

```
[ndb_mgmd]
```

```
hostname=192.168.0.183
```

```
datadir= /var/lib/mysql-cluster
```

```
[ndbd]
```

```
hostname=192.168.0.12
```

```
[ndbd]
```

```
hostname=192.168.0.188
```

```
[mysqld]
```

```
hostname=192.168.0.183
```

这个例子给出了一个简单的双节点集群复制所需的最少变量。因此，NoOfReplicas 选项设置为 2。注意，我们将 datadir 设为 /var/lib/mysql-cluster，也可以随便设置，但大多数 MySQL 集群都采用这个目录。

最后，注意我们给每个节点都指定了一个主机名，这很重要，因为 NDB 管理守护进程需要知道集群中所有节点的位置。如果你已经下载并安装了 MySQL 集群，接下来就是对主机名做必要的更改，以配合我们的例子。

MySQL 集群的配置文件默认放在 /var/lib/mysql-cluster 目录下，并命名为 config.ini。



数据节点上并不需要完整安装 MySQL 集群二进制包。后面你就会发现，只需要在数据节点上安装 ndbd 守护进程即可。



## 启动 MySQL 集群

启动 MySQL 集群需要有序地执行一组命令。我们就用上面的例子逐步跟踪启动集群过程，不过首先简单看看这个过程的一般步骤：

1. 启动管理节点。
2. 启动数据节点。
3. 启动 MySQL 服务器（SQL 节点）。

本例中，我们先在 192.168.0.183 上启动 NDB 管理节点，然后分别在 192.168.0.12 和 192.168.0.188 上启动每个数据节点。数据节点开始运行后，在 192.168.0.183 上启动 MySQL 服务器。经过一段简短的启动延迟之后，集群就可以使用了。

### 启动管理节点

第一个要启动的节点是名为 *ndb\_mgmd* 的 NDB 管理守护进程，位于 MySQL 安装目录的 *libexec* 文件夹下。例如，它在 Ubuntu 上的位置为 */usr/local/mysql/libexec*。

通过超级用户启动 NDB 管理守护进程，并指定 *--initial* 和 *--config-file* 选项。*--initial* 选项告诉集群这是第一次启动，需要清除以前启动时存储的配置信息。*--config-file* 选项告诉守护进程配置文件的位置。示例 9-3 显示了如何启动 NDB 管理守护进程。

309

示例9-3：启动NDB管理守护进程

```
$ sudo ../libexec/ndb_mgmd --initial \
--config-file /var/lib/mysql-cluster/config.ini
MySQL Cluster Management Server mysql-5.6.11 ndb-7.3.2
```

启动时最好提供 *--config-file* 选项，因为有些安装中的配置文件搜索模式具有不同的默认位置。使用命令 *ndb\_mgmd --help*，并查找“Default options are read from”，就可以查看其默认安装位置。而以后的守护进程启动就不需要再指定 *--config-file* 选项了。

### 启动管理控制台

现在启动 NDB 管理控制台，并检查 NDB 管理守护进程是否正确地读取配置。虽然这一步不是必需的，但最好这么做。NDB 管理控制台的名字为 *ndb\_mgm*，位于 MySQL 安装目录的 *bin* 目录下。使用 *SHOW* 命令可以查看配置，如示例 9-4 所示。

示例9-4：初始启动NDB管理控制台

```
$ ./ndb_mgm
-- NDB Cluster -- Management Client --
ndb_mgm> SHOW
Connected to Management Server at: 192.168.0.183:1186
```

## Cluster Configuration

```
-----  
[NDBd(NDB)]          2 node(s)  
id=2 (not connected, accepting connect from 192.168.0.188)  
id=3 (not connected, accepting connect from 192.168.0.12)  
  
[NDB_mgmd(MGM)]      1 node(s)  
id=1 @192.168.0.183 (mysql-5.5.31 ndb-7.2.13)  
  
[mysqld(API)]        1 node(s)  
id=4 (not connected, accepting connect from 192.168.0.183)  
  
ndb_mgm>
```

这个命令显示了数据节点及其 IP 地址，以及 NDB 管理守护进程和 SQL 节点。这时检查所有节点的 IP 地址是否配置正确，以及所有数据节点是否正确装载。如果更改了集群配置却在这里看到旧值，可能是 NDB 管理控制台还没有读取到新的配置文件。

**310** 这个输出告诉我们 NDB 管理控制台已经装载成功并准备就绪。如果还没有，SHOW 命令会出现通信错误而失败。如果出现这个错误，首先检查 NDB 管理客户端与 NDB 管理守护进程是否运行在同一个服务器上。如果不是，使用 `--ndb-connectstring` 选项，并提供 IP 地址或 NDB 管理守护进程所在机器的主机名作为参数。

最后，注意节点的节点 ID。从 NDB 管理控制台发出命令指定集群中的节点时，需要这个信息。任何时候使用 HELP 命令都可以查看其他可用的命令信息。还需要知道 SQL 节点的节点 ID，这样可以正确地启动它们。



在文件中使用 `--ndb-nodeid` 参数指定集群中每个节点的节点 ID。

还可以用 STATUS 命令查看节点状态，ALL STATUS 命令可以查看所有节点的状态，`node-id STATUS` 查看特定节点的状态。这个命令用于监视集群的启动，因为输出了数据节点正处于哪个启动阶段。关于数据节点启动阶段的更多细节请参考在线 MySQL 参考手册的 MySQL 集群部分。

## 启动数据节点

既然我们已经启动了 NDB 管理守护进程，现在该启动数据节点了。但是，在这之前，让我们先看看创建一个 NDB 数据节点的最小配置需求。

为了创建 NDB 数据节点，你需要为目标宿主操作系统编制一个 NDB 数据节点守护进程（即 *ndbd*）。首先，创建文件夹 */var/lib/mysql-cluster*，然后将 *ndbd* 可执行文件复制进来，完成！显然，这样很容易通过脚本实现数据节点的创建（很多数据节点都有这样的脚本）。

可以使用 *--initial-start* 选项启动数据节点（*ndbd*），表明这是第一次启动集群。还必须提供 *--ndb-connectstring* 选项，加上 NDB 管理守护进程的 IP 地址作为参数。示例 9-5 给出了第一次启动数据节点的示例，每个节点都执行这样的操作。

示例9-5：启动数据节点

```
$ sudo ./ndbd --initial-start --ndb-connectstring=192.168.0.183
2013-02-11 06:22:52 [ndbd] INFO -- Angel connected to '192.168.0.183:1186'
2013-02-11 06:22:52 [ndbd] INFO -- Angel allocated nodeid: 2
```

如果你正在启动新的数据节点，需要重置数据节点，或者进行故障恢复，可以指定 *--initial* 选项来强制数据节点清除已有的配置信息和缓存数据，然后从 NDB 管理守护进程请求一份新的副本。

311



小心使用 *--initial* 选项，它们真的会删除你的数据！

返回管理控制台，然后检查状态，如示例 9-6 所示。

示例9-6：数据节点的状态

```
ndb_mgm> SHOW
Cluster Configuration
-----
[ndbd(NDB)] 2 node(s)
id=2 @192.168.0.188 (mysql-5.5.31 ndb-7.2.13, Nodegroup: 0, Master)
id=3 @192.168.0.12 (mysql-5.5.31 ndb-7.2.13, Nodegroup: 0, Master)

[ndb_mgmd(MGM)] 1 node(s)
id=1 @192.168.0.183 (mysql-5.5.31 ndb-7.2.13)

[mysqld(API)] 1 node(s)
id=4 (not connected, accepting connect from 192.168.0.183)
```

可以看到数据节点已经成功启动，因为已经显示了守护进程信息。还可以发现其中一个节点被选为集群复制的 master。因为配置文件中副本的数目设为 2，所以将会有两份数据副本。不要将这里的 master 概念与 MySQL 复制中的 master 混淆，后面将详细讨论两者的差别。

## 启动 SQL 节点

一旦数据节点开始运行，就可以连接 SQL 节点。要使 MySQL 服务器连接到 NDB 集群，必须指定几个选项。大部分时候在 *my.cnf* 文件中指定这些选项，当然也可以在启动服务器时用命令行指定。

**ndbcluster**

告诉服务器想使用哪一个 NDB 集群存储引擎。

**ndb\_connectstring**

告诉服务器 NDB 管理守护进程的位置。

312 **ndb\_nodeid** 和 **server\_id**

一般设为节点 ID，可以在管理控制台中用 **SHOW** 命令输出节点 ID 信息。

示例 9-7 给出了集群中 SQL 节点的正确启动顺序。

示例9-7: 启动SQL节点

```
$ sudo ../libexec/mysqld --ndbcluster \
--console -umysql
130211 9:14:21 [Note] Plugin 'FEDERATED' is disabled.
130211 9:14:21 InnoDB: Started; log sequence number 0 1112278176
130211 9:14:21 [Note] NDB: NodeID is 4, management server '192.168.0.183:1186'
130211 9:14:22 [Note] NDB[0]: NodeID: 4, all storage nodes connected
130211 9:14:22 [Note] Starting Cluster Binlog Thread
130211 9:14:22 [Note] Event Scheduler: Loaded 0 events
130211 9:14:23 [Note] NDB: Creating mysql.NDB_schema
130211 9:14:23 [Note] NDB: Flushing mysql.NDB_schema
130211 9:14:23 [Note] NDB Binlog: CREATE TABLE Event: REPL$mysql/NDB_schema
130211 9:14:23 [Note] NDB Binlog: logging ./mysql/NDB_schema (UPDATED,USE_WRITE)
130211 9:14:23 [Note] NDB: Creating mysql.NDB_apply_status
130211 9:14:23 [Note] NDB: Flushing mysql.NDB_apply_status
130211 9:14:23 [Note] NDB Binlog: CREATE TABLE Event: REPL$mysql/NDB_apply_status
130211 9:14:23 [Note] NDB Binlog: logging ./mysql/NDB_apply_status
(UPDATED,USE_WRITE)
2013-02-11 09:14:23 [NdbApi] INFO      -- Flushing incomplete GCI:s < 65/17
2013-02-11 09:14:23 [NdbApi] INFO      -- Flushing incomplete GCI:s < 65/17
130211 9:14:23 [Note] NDB Binlog: starting log at epoch 65/17
130211 9:14:23 [Note] NDB Binlog: NDB tables writable
130211 9:14:23 [Note] ../libexec/mysqld: ready for connections.
Version: '5.5.31-ndb-7.2.13-cluster-gpl-log' socket: '/var/lib/mysql/mysqld.sock'
port: 3306 Source distribution
```

输出中包括了关于 NDB 集群连接、日志和状态等额外的注释信息。如果没有看到注释，



或是发现了错误，首先检查 SQL 节点的启动选项是否正确。节点 ID 和管理服务器的相关消息特别重要。如果有多个管理服务器正在运行，那么要保证 SQL 节点与正确的服务器通信。

SQL 节点正确启动之后，返回管理控制台，检查所有节点的状态，如示例 9-8 所示。

示例9-8: 正在运行的集群的状态示例

```
ndb_mgm> SHOW
Cluster Configuration
-----
[NDBd(NDB)] 2 node(s)
id=2 @192.168.0.188 (mysql-5.5.31 ndb-7.2.13, Nodegroup: 0, Master)
id=3 @192.168.0.12 (mysql-5.5.31 ndb-7.2.13, Nodegroup: 0)

[NDB_mgmd(MGM)] 1 node(s)
id=1 @192.168.0.183 (mysql-5.5.31 ndb-7.2.13)

[mysqld(API)] 1 node(s)
id=4 @192.168.0.183 (mysql-5.5.31 ndb-7.2.13)
```

313

所有节点都已经连接上且正常运行。如果除了这里显示的信息你还看到了其他信息，那么你在启动节点的过程中就遇到了问题。检查每个节点的日志信息，确定哪里出错。最常见的问题是网络连接故障（如防火墙等）。NDB 节点默认使用端口 1186。

数据节点和 NDB 管理守护进程的日志文件位于数据目录下，SQL 节点的日志则位于 MySQL 服务器的常规位置。

## 测试集群

我们的例子中的集群已经在运行了，下面进行一个简单的测试，如示例 9-9 所示，以确保可以使用 NDB 集群存储引擎创建数据库和表。

示例9-9: 测试集群

```
mysql> create database cluster_db;
Query OK, 1 row affected (0.06 sec)

mysql> create table cluster_db.t1 (a int) engine=NDBCLUSTER;
Query OK, 0 rows affected (0.31 sec)

mysql> show create table cluster_db.t1 \G
***** 1. row *****
      Table: t1
Create Table: CREATE TABLE `t1` (
```

```

    `a` int(11) DEFAULT NULL
) ENGINE=NDBcluster DEFAULT CHARSET=latin1
1 row in set (0.00 sec)

mysql> insert into cluster_db.t1 VALUES (1), (100), (1000);
Query OK, 3 rows affected (0.00 sec)
Records: 3 Duplicates: 0 Warnings: 0

mysql> select * from cluster_db.t1 \G
***** 1. row *****
a: 1
***** 2. row *****
a: 1000
***** 3. row *****
a: 100
3 rows in set (0.00 sec)

```

**314** 集群已经在运行中，可以通过装载数据和运行示例查询来进行试验。在数据更新过程中，让其中一个数据节点失效，然后重启它，看看单个数据节点的故障会不会影响整个系统的可访问性。

## 关闭集群

正如启动是有特定顺序的，关闭集群同样也要按照一定的顺序进行：

1. 如果集群间有复制正在运行，先使 slave 跟上进度，然后再停止复制。
2. 关闭 SQL 节点 (*mysqld*)。
3. 在 NDB 管理控制台中发出 SHUTDOWN 命令。
4. 退出 NDB 管理控制台。

如果 MySQL 复制运行在两个或更多的集群之间，首先要保证在关闭 SQL 节点之前，复制 slave 赶上了 master 的进度（即同步）。

在 NDB 管理控制台中发出 SHUTDOWN 命令，将会关闭所有的数据节点和 NDB 管理守护进程。

## 获得高可用性

使用高可用性的主要目的是保持服务可用。对于数据库系统来说，这就意味着我们必须保证数据总是可访问的。MySQL 集群就是用来满足这个需要的。MySQL 集群通过以下方式支持高可用性：数据节点之间的数据分布（减少单个节点的数据丢失风险），集群中副本之间的复制，丢失的数据节点的自动恢复（故障转移），通过心跳进行数据故障检测，

以及使用本地和全局检查点来保证数据一致性等。

让我们先看看高可用性数据库系统的一些特点。高可用性的数据库系统（或任何系统）必须满足以下要求：

- 99.999% 的上线率
- 无单点故障
- 故障转移
- 容错

99.999% 的上线时间（uptime）意味着从实践的角度来说，数据总是可用的，即数据库服务器提供不间断的、连续的服务。假设前提是服务器永远不会由于组件故障或维护问题而离线。所有的操作，如维护和恢复，都是联机进行的，访问不会被中断。

◀ 315

这种理想情况的需求很少，只有最关键的行业才真正要求这样的质量。此外，还需要短期的例程序及预防性的维护工作（从而达到将近 100% 的上线率）。有趣的是，关于可接受的上线时间的粒度是以上线率中 9 的个数来衡量的。表 9-1 给出了可接受的宕机时间（离线时间）上线率等级（单位：年）。

表9-1：可接受的宕机时间表

上线时间	可接受的宕机时间
99.000%	3.65 天
99.900%	8.76 小时
99.990%	52.56 分
99.999%	5.26 分

注意，从这个表中可以看出，级别中的 9 数目越多，可接受的宕机时间就越短。例如 99.999% 的上线率，意味着该系统在 1 年时间内，除了极少时间外，都要不间断地联机执行所有的维护操作。MySQL 集群可以通过很多种方式满足这种需求，包括数据节点轮流重启的能力、数据库的联机维护操作，以及多种数据访问渠道（通过 NDB API 连接的 SQL 节点和应用程序）等。

无单点故障意味着系统中不存在这样的单个组件，它能够决定整个服务的可用性。通过为集群中每种类型的节点设置冗余，可以做到这一点。在前面的例子中，我们有两个数据节点，因此可以防止单个数据节点故障。但是，我们只有一个管理节点和一个 SQL 节点。理想情况下，还需要为这些功能添加额外的节点。MySQL 集群支持多个 SQL 节点，这样即使管理节点失效，集群仍然可以运行。

故障转移是指，如果一个组件失效，另一个组件可以代替它完成同样的功能。对 MySQL

数据节点来说,如果集群中包含多个数据副本,故障转移就可以自动执行。如果某个副本的一个 MySQL 数据节点失效,数据访问不会被中断。重启丢失的数据节点,它将从另一个副本中把数据复制回来。对 SQL 节点来说,由于数据实际是存储在数据节点上的,任何 SQL 节点之间都可以相互代替。

如果 NDB 管理节点发生故障,集群仍然可以继续运行,任何时候你都可以重新启动一个新的管理节点(如果配置没有发生改变的话)。

316 而且还可以使用前面章节中讨论的常规高可用性解决方案,包括整个集群之间的复制和自动故障转移。本章后面的部分将更详细地讨论集群复制。

容错一般与硬件相关,如备用电源、冗余的网络渠道等。对于软件系统,容错则是处理故障转移的附带结果。对 MySQL 集群来说,容错意味着允许一定数量的故障发生,而且能够继续访问数据。就像硬件 RAID 系统在同一个 RAID 阵列丢失两个驱动一样,副本之间丢失多个数据节点,会产生不可恢复的故障。但是,经过谨慎规划,可以配置 MySQL 集群以减小这种风险性。适当的监控和主动维护也可以减小风险。

MySQL 集群可以通过主动管理集群中的节点来达到容错。MySQL 集群使用心跳来检查服务是否可用,一旦发现节点故障,就会执行恢复。

MySQL 集群中的日志机制还提供了故障转移和容错的恢复级别。本地和全局检查点保证了集群中的数据是一致的。该信息对于数据节点故障的快速恢复非常重要,不仅能够恢复数据,检查点的唯一性还允许快速恢复节点。后面会详细讨论这个特性。

图 9-5 描绘了 Web 服务场景下 MySQL 集群的高可用性配置。

图中的虚线框表示系统边界。为了保证冗余,这些组件都处于独立的硬件上,而且,要将 4 个数据节点配置为 2 个副本。这幅图中没有显示与应用程序交互的附加组件,例如用来分离 Web 和 MySQL 服务器负载的负载均衡器。

在配置高可用性 MySQL 集群时,应该考虑使用下面的最佳实践(在后面考察高性能 MySQL 集群技术的时候,将进一步讨论这个问题)。

- 在不同硬件的数据节点上使用多个副本。
- 使用冗余的网络连接以防止网络出现故障。
- 使用多个 SQL 节点。
- 使用多个数据节点来提高性能,并将数据分布化。



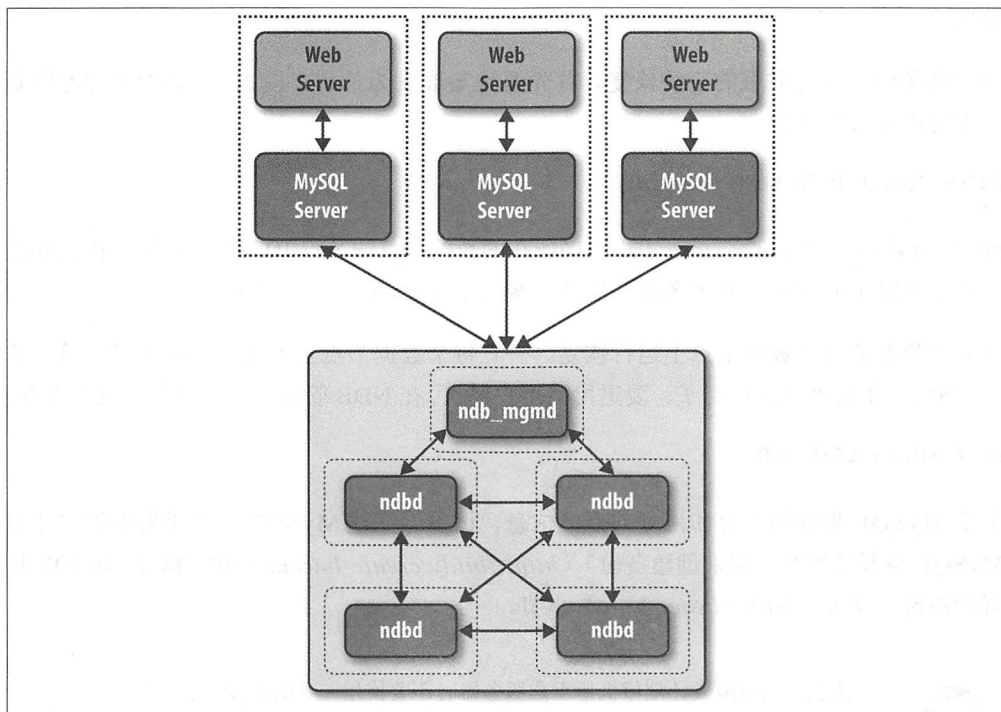


图9-5：一个高可用MySQL集群

## 系统恢复

317

有两种类型的系统恢复，一种是关闭服务器做维护或类似的计划性事件，另一种是非预期的系统处理能力丢失。幸好，MySQL 集群提供了一种恢复功能的机制，即使最坏的情况发生。

如果 MySQL 集群被正确关闭，它会从日志中的检查点进行恢复。这类似一个自动的常规启动阶段。系统会从每个数据节点的本地检查点装载最新数据，从而在重启时将数据恢复到最新的快照。一旦数据节点从本地检查点装载了数据，系统就会执行重做日志，以达到最近的全局检查点，从而将数据同步到系统关闭前的最后变更位置。无论是有意关机后的重启，还是故障后的完全系统重启，这个过程都是一样的。

也许你认为启动是不能“恢复”的，但记住，MySQL 集群是一个内存数据库，因此，数据必须在启动时从磁盘重新装载。这个过程通过装载数据至最近的检查点来完成。

从重大灾难中恢复系统或进行纠正性措施时，还需要从数据的备份中恢复。前面提过，

318

数据。

为了从备份中执行完整的系统恢复，首先要将集群设为单用户模式，在 NDB 管理控制台中使用下面的命令：

```
ENTER SINGLE USER MODE node-id
```

*node-id* 是指调用 *ndb\_restore* 实用程序所在的数据节点的节点 ID。关于单用户模式和连接基于 API 的实用程序的更多详细信息，参见在线 MySQL 参考手册。

然后在集群的每个数据节点上运行恢复。一旦每个数据节点上的数据都恢复了，退出单用户模式，集群便可以使用了。要退出单用户模式，在 NDB 管理控制台中发出以下命令：

```
EXIT SINGLE USER MODE
```

关于 MySQL 集群的备份和恢复的更多信息，请参见在线 MySQL 参考手册中的“使用 MySQL 集群管理客户端来创建备份”(<http://bit.ly/create-backup>)和“恢复 MySQL 集群的备份”(<http://bit.ly/cluster-bkup>)小节。



注意，在故障或计划停机后重启服务器时不要使用 `--initial` 选项。

## 节点恢复

节点发生故障的原因有很多，包括网络、硬件、内存或操作系统的问题或故障。这里讨论导致这些故障最常见的原因，以及 MySQL 集群是如何处理节点恢复的。本节中重点考虑数据节点，因为它们是保证数据可访问的重要节点。

### 硬件

当主机硬件发生故障时，显然运行在那个系统上的数据节点也会失效。这种情况下，MySQL 集群会将故障转移到其他副本上。要对这种故障进行恢复，需要更换发生故障的硬件，然后重启数据节点。

### 网络

如果数据节点所在的网络由于某种网络硬件或软件故障而变得不可用，节点或许可以继续执行，但由于它无法与其他节点取得联系（通过心跳），MySQL 集群会将该节点标记为“宕机”，然后将故障转移到其他副本，直到该节点恢复。恢复这种故障需要更换发生故障的网络硬件，然后重启数据节点。

## 内存

如果主机操作系统的内存不足,集群就会出现数据空间不足。这会导致数据节点失效。为了解决这个问题,可以增加更多的内存,或者增加配置参数的内存分配值,然后执行数据节点的滚动重启。

## 操作系统

如果操作系统的配置干扰了数据节点的执行,就解决问题,然后重启数据节点。

关于数据库高可用性和 MySQL 高可用性的更多信息,参见 MySQL 网站上的白皮书。

## 复制

我们已经简单讨论了 MySQL 复制及集群内部的复制之间的区别。MySQL 集群复制有时又称内部集群复制 (internal cluster replication) 或简称为内部复制 (internal replication),以表明它不是 MySQL 复制。MySQL 复制有时又称外部复制 (external replication)。

本节将讨论 MySQL 集群内部复制,以及 MySQL 复制(即外部复制)在 MySQL 集群之间(而不是单个 MySQL 服务器之间)是如何进行数据复制的。

### 集群内部复制和 MySQL 复制

前面讲过,MySQL 集群内部使用同步复制,支持两阶段提交协议,保证数据的完整性。相反,MySQL 复制使用异步复制,即依赖于稳定交付的单向数据传输,而且在数据提交之前不需要确认。

### 集群内部复制

内部 MySQL 集群复制通过存储数据的多个副本来提供冗余。这个过程保证在查询被确认完成(即提交)之前,数据被写入多个节点。这是通过两阶段提交实现的。

复制的形式是同步的,因为要保证在查询确认或提交完成时数据是一致的。

数据以片段(fragment)的形式复制,这里片段是指表中行的一个子集。片段由于分区(partitioning)在各个数据节点之间分布开来,而且在每个副本的其他数据节点之上都有该片段的一份副本。其中一个片段将作为主副本,用来执行查询。所有其他的相同数据副本都作为备用片段。数据更新时,主片段会首先得到更新。

320

## 集群之间的 MySQL 复制

集群之间的复制很简单。如果你能够在两个 MySQL 服务器之间建立复制，那么就可以在两个 MySQL 集群之间建立复制。因为启动集群之间的复制并没有什么特殊的设置步骤或者额外的命令或参数。MySQL 复制就像在单个服务器之间那样工作，只不过数据存储在 NDB 集群中。但是，外部复制有一些限制。这里列出了规划外部复制需要考虑的一些问题（如果想知道更多关于外部复制的信息，请参考在线 MySQL 参考手册的“MySQL 集群复制”一节）。

- 外部复制必须是基于行的。
- 外部复制不能是环形的。
- 外部复制不支持 `auto_increment_*` 选项。
- 二进制日志的大小可能比常规的 MySQL 复制更大。

MySQL 复制将数据从一个集群复制到另一个集群，可以利用每个站点上 MySQL 集群的优势将数据复制到其他站点。

### MySQL 复制可以在 MySQL 集群中使用吗

可以从一个 MySQL 集群服务器复制到另一个非 MySQL 集群服务器，反之亦然）。除了要考虑到一些潜在的存储引擎冲突以外，不需要做任何特殊配置，就像在使用不同存储引擎的 MySQL 服务器之间进行复制一样。这种情况下，也可以使用默认的存储引擎，而不用在 CREATE 语句中指定存储引擎。

从一个 MySQL 集群复制到一个非 MySQL 集群需要创建名为 `ndb_apply_status` 的特殊的表，来复制提交的时间。如果 slave 上的这张表丢失，复制就会报 `ndb_apply_status` 不存在的错误而停止复制。可以使用如下命令创建这个表：

```
CREATE TABLE `mysql`.`ndb_apply_status` (  
  `server_id` INT(10) UNSIGNED NOT NULL,  
  `epoch` BIGINT(20) UNSIGNED NOT NULL,  
  `log_name` VARCHAR(255) CHARACTER SET latin1  
  COLLATE latin1_bin NOT NULL,  
  `start_pos` BIGINT(20) UNSIGNED NOT NULL,  
  `end_pos` BIGINT(20) UNSIGNED NOT NULL,  
  PRIMARY KEY (`server_id`) USING HASH  
) ENGINE=NDBCLUSTER DEFAULT CHARSET=latin1;
```

使用外部复制的 MySQL 集群复制需要采用基于行的 MySQL 复制，主 SQL 节点必须用 `--binlog-format=ROW` 或 `--binlog-format=MIXED` 启动。MySQL 复制的所有其他需求也



都要满足（例如所有 SQL 节点都要有唯一的服务器 ID）。

外部复制的复制过程还需要一些特别操作，包括使用集群二进制日志，binlog 注入线程和特殊的系统表来支持集群之间的更新等。外部复制在处理事务型变更时也有所不同，这一点我们将在下一节中详细讨论。

## MySQL 集群（外部）复制的架构

可以认为外部复制与 MySQL 复制操作的基本概念是相同的。具体来说，某个集群安装时需要定义 master 和 slave，这样，master 包含数据的原始副本，而 slave 基于数据变更流增量地接收数据的副本。

MySQL 集群复制使用 *mysql* 数据库中的专用表，它们位于 master 和 slave（无论 slave 是单个服务器还是一个集群）的每个 SQL 节点上。这些表在 MySQL 安装过程中被创建，包括：*ndb\_binlog\_index* 表存储二进制日志（本地 SQL 节点的）的索引数据，*ndb\_apply\_status* 表存储已经复制到 slave 上的操作记录。*ndb\_apply\_status* 表在所有 SQL 节点上维护，并保持同步，从而保证该表在整个集群中相同。这个表可用于对 MySQL 集群中失效的复制 slave 进行即时恢复（PITR）。

这些表由一个被称为 binlog 注入线程（binlog injector thread）的新线程更新，这个线程通过记录集群中的变更来保证 NDB 集群存储引擎的任何变更都能在 master 上更新。binlog 注入线程负责根据二进制日志中的记录来获取集群中所有的数据事件，并保证所有更改、插入或删除数据的事件都被写入 *ndb\_binlog\_index* 表中。master 的转储（dump）线程使用 MySQL 复制将这些事件发给 slave 的 I/O 线程。

外部复制与 MySQL 复制的一个重要不同之处在于，每个 epoch 都被视为一个事务。因为 epoch 是指每两个检查点之间的时间跨度，MySQL 集群保证每个检查点的一致性，所以 epoch 是原子性的，像 MySQL 复制中的事务那样被复制。关于上一次应用的 epoch 信息存储在 NDB 系统表中，支持 MySQL 集群之间的外部复制。

322

## 单通道复制和多通道复制

master 和 slave 之间的 MySQL 复制连接称为一个通道（channel）。实际上通道是指 master 连接 slave 所使用的网络协议和媒介。一般只有单通道，但为了保证最大可用性，可以建立一个备用通道用来容错，称为多通道复制。多通道外部复制如图 9-6 所示。

多通道复制大大提高了网络连接故障的恢复能力。理想情况下，使用主动监控机制触发潜在的网络连接故障，以告知连接何时失效。该过程的实现方法有很多，如使用简单心跳机制编写脚本来报警，以及使用 MySQL 企业监控器（MySQL Enterprise Monitor）中的顾问等。

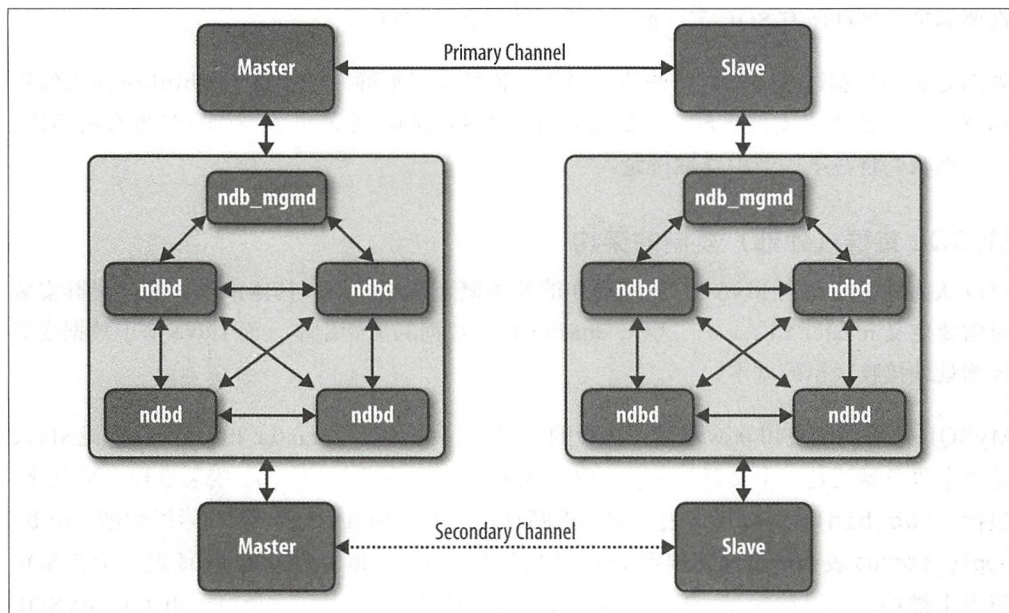


图9-6: 多通道外部复制

注意，图 9-6 中一共有 4 个 SQL 节点（即 MySQL 服务器）。其中 master 的集群有 2 个 master 节点（即 SQL 节点），一个主（primary），一个备用（secondary）。同样，slave 集群中也有一个主 slave 和一个备用 slave。主 master/slave 之间通过一个网络连接通信，备用 master/slave 使用另一个网络连接通信。



不要想当然地认为网络组件是可靠的，哪怕一个交换器也可能会失效。同一个交换网络上使用不同布线起不了什么作用。最好是使用完全独立的冗余连接和中介网络组件，以达到真正的网络冗余。

多通道复制的建立与单通道（常规）MySQL 复制没什么不同。然而，复制的故障转移略有不同，其思想是无须在备用通道上启动 slave。故障转移到备用通道上需要一些特别的步骤。

按照下面的步骤来启动多通道外部复制，激活主通道，使备用通道处于待机模式。（假定冗余网络通信和硬件都已准备就绪并运行良好。）

1. 启动主 master。
2. 启动备用 master。
3. 将主 slave 连接到主 master。

4. 将备用 slave 连接到备用 master。
5. 启动主 slave。



不要启动备用 slave (使用 `START SLAVE`), 否则可能会造成主键冲突和数据重复的问题。但是, 要为备用 slave 配置备用 master 的信息 (使用 `CHANGE MASTER`), 这样如果主通道发生故障, 备用通道才能快速启动。

故障转移到备用通道的步骤则不同。仅仅启动备用 slave 是不够的。为了避免同样的数据被复制两次, 首先要确定上一次复制的 epoch, 然后用它启动复制过程。这个过程如下 (注意使用变量保存中间结果):

1. 找到 slave 接收到的最近全局检查点的时间, 这需要从主 slave 的 `ndb_apply_status` 表中找到最近的 epoch:

```
SELECT @latest := MAX(epoch) FROM mysql.ndb_apply_status;
```

2. 获取故障后出现在主 master 上的 `ndb_binlog_index` 表中的行, 可以用下面的查询 ◀ 324  
从主 master 中找到这些行:

```
SELECT @file := SUBSTRING_INDEX(File, '/', -1), @pos := Position
FROM mysql.ndb_binlog_index
WHERE epoch > @latest ORDER BY ASC LIMIT 1;
```

3. 同步备用通道。在备用 slave 上运行下面的命令, 这里 `file` 是实际文件名, `pos` 是位置:

```
CHANGE MASTER TO MASTER_LOG_FILE = 'file', MASTER_LOG_POS = pos;
```

4. 在备用通道上启动复制, 在备用 slave 上运行这个命令:

```
START SLAVE;
```

这个故障转移过程会切换复制通道。如果 SQL 节点发生故障, 则必须在执行这个故障转移过程之前处理并修复问题。



最好保证主通道确实是离线的。为了以防万一还要考虑停止主 slave。

## 获得高性能

MySQL 集群并不仅仅是为了高可用性而设计的, 还保证了高性能。我们已经讨论了很多高可用性的特性。本节将考察提供高性能的一些特性。这里总结了一些获得系统高性能的最佳实践列表。



下面这些特性支持 MySQL 集群的高性能，其中有些前面章节已经提到过：

#### 集群间复制（全局冗余）

所有数据都复制到一个远程站点，这个远程站点可以卸载（offload）主站点。

#### 集群内部复制（本地冗余）

多个数据节点可以并行读取数据。

#### 主存储器存储

无须等待磁盘写，保证了数据更新的快速处理。

325

## 高性能的注意事项

使系统支持高性能有 3 个重要的注意事项。

- 保证应用程序尽可能高效。有时候需要更改数据库服务器（如优化服务器配置或修改数据库模式），而应用程序则通常可以被设计或重构成更高性能的应用。
- 最大化数据库的访问。包括提供足够的 MySQL 服务器连接（即横向扩展），以及为了可用性而分发数据，如通过复制的方式。
- 考虑将你的 MySQL 集群性能提高，例如，增加更多的数据节点。



JOIN 查询通常很费时。主要是因为 MySQL 集群的分布式特性，而且 MySQL 服务器并不能很好地支持 MySQL 集群的 JOIN 查询。在 MySQL 集群 7.2 之前，JOIN 操作是通过从数据节点获取数据，然后在 SQL 节点内部执行连接完成的，需要的数据通过网络来回传输。在 MySQL 集群 7.2 中，MySQL 服务器 5.5 支持将连接操作放到引擎中执行，这就减少了通过网络传输的数据量，而且在多个数据节点上执行连接也能够提高并行性。

或许需要在期望的高可用性等级和高性能之间做出权衡。例如，通过添加更多的副本来增加可用性。但是，用来防止数据节点丢失的副本越多，就需要越强大的处理能力，而且更新过程的性能也会降低。数据读取依然很快，因为相同数据的读取并不需要多个副本。保持较少的副本和较多的数据节点（即横向扩展），可以获得更高的写性能。

另一个重要的注意事项是 MySQL 集群的分布式特性。因为每个节点在独立服务器上运行时效果最好，所以每个服务器的性能很重要，但网络组件也同样重要。由于协调性命令和数据在节点之间传输，网络的互联性能必须也要适应高性能。还要考虑一些参数，如端口选择（如 TCP/IP、SHM 和 SCI）、延迟、带宽和地理上的接近度。

326

可以在云环境中搭建和运行 MySQL 集群。这样做的好处之一就是网络互联速度很快，而且是优化的。由于数据节点需要快速的处理器、足够的内存和快速的网络，所以在云环境中使用虚拟服务器技术构建 MySQL 集群绰绰有余。但是要注意，MySQL 集群官方



并不支持虚拟服务器环境。

从在线 MySQL 参考手册的“MySQL 集群”一节中可以找到完整的注意事项列表。关于 MySQL 性能提升，可以参考 *High Performance MySQL* 一书。

## 高性能的最佳实践

保证 MySQL 集群运行的性能最佳有很多办法。这里列出了一些最常用的性能提升实践，每项附有简要的讨论。其中有些较为常见，但是在讨论高性能的时候我们并不想忽略它们。

### 调整访问模式

考虑应用程序访问数据的方法。由于 MySQL 集群在内存中存储索引字段，所以通过索引字段而不是非索引字段访问数据带来的性能提升，比单个 MySQL 服务器多得多。MySQL 集群要求每张表都有主键，所以根据主键检索数据的应用程序几乎都很快。

### 确保你的应用程序是分布感知的（distribution-aware）

在已分区的数据存储上访问数据时，最好将查询局限在集群中的单个节点上。默认情况下，MySQL 集群使用主键将行哈希到各个分区。但不幸的是，如果考虑到主表 / 细节表（master/detail）的查询（通常应用程序先访问主表，然后从细节表获取细节信息，该细节表参照了主表），这样做并不总是最佳的。在这种情况下，就要修改哈希函数，以保证主表中的行和细节表中的行位于同一个节点上。一种实现方式是分区裁剪（partition pruning），去掉细节表哈希分区时使用的字段，仅根据主表的主键（即细节表的外键）对细节表中的行进行分区。这样主表和细节表中的行就被分配到分区树（partition tree）中的同一个节点上了。

### 使用批操作

每个查询的往返都具有显著的开销。像插入这样的操作，可以使用一个多条插入查询（一次插入多行的 INSERT 语句）来减少开销。启用 `transaction_allow_batching` 参数，并在单个事务（BEGIN 和 END 之间的语句块）中包含多个操作，也可以进行批量操作。这样你可以做多个数据操作查询（INSERT、UPDATE 等），并减少开销。



`transaction_allow_batching` 选项不能在带变量的 SELECT 语句和 UPDATE 语句中使用。

327

## 优化数据库模式

MySQL 集群中的数据库模式优化与一般的数据系统一样。MySQL 集群使用高效的数据类型（如节省内存所需的最小规模；一张百万行记录的表每行 30 字节，可以节省大量的内存）。为了利用 MySQL 集群的并行数据访问方法（分区），还要考虑将某些模式非规范化（denormalization）。

## 优化查询

显然，查询被优化得越多，查询性能就越高。对所有数据库来说都是这样，而首先要做的就是提高应用程序的性能。对 MySQL 集群来说，从数据检索的角度考虑查询优化。MySQL 集群的性能对 JOIN 操作尤其敏感。性能很差的查询有时可能导致异常，容易被误认为是系统其他部分的低效所致。

## 优化服务器参数

优化集群配置可以保证其运行尽可能高效。这意味着要花些时间理解很多配置选项，而且要确定使用正确的硬件。这项任务没有任何神奇的地方——参数修改得越多，你的安装设置就越独一无二。这项实践要谨慎使用，一次只修改一个参数，并且总要在改变之前与已知的基准进行比较。

## 使用连接池

默认情况下，SQL 节点仅使用单个线程连接 NDB 集群。如果有更多的连接线程，SQL 节点可以一次执行多个查询。要想在 SQL 节点中使用连接池，需要向配置文件中添加 `ndb-cluster-connection-pool` 选项。将值设为大于 1 的数（比如 4），并把这个选项放在 `[mysqld]` 配置段中。这个设置需要试验，因为对于应用或硬件来说，这个值通常太高了。

## 使用多线程数据节点

如果数据节点有多核 CPU 或多个 CPU，运行名为 `ndbmt` 的多线程数据节点守护进程，就可以获得额外的性能提升。这个守护进程能够使用高达 8 个 CPU 核或线程。多个线程使得数据节点可以并行执行多个操作，如本地查询处理器（Local Query Handler, LQH）和通信进程，以获取更高的吞吐量。

## 使用 NDB API 个性化应用程序

MySQL 服务器（即 SQL 节点）提供了一个快速的查询处理器前端，而 MySQL 构建了一种直接访问 C++ 的机制，称为 NDB API。对某些操作来说，如 MySQL 集群与 LDAP 的接口，这可能是将 MySQL 集群（这里指的是 NDB 集群）连接到你的应用程序的唯一办法。如果对应用程序的性能要求很高，而且你具备个性化 NDB API 方案所需的开发资源，你会发现性能将得到大大提升。

## 使用正确的硬件

一般来说,硬件越快,性能就越好。但是,要考虑到集群配置的各个方面。不仅要有更快的 CPU 和更大更快的内存,还要有诸如 SCI 之类的高速网络互联,以及高速的硬件冗余的网络连接。很多情况下,这些硬件方案被构建成为可立即投入使用的商品,而不用重新配置集群。

## 不要使用交换空间

确保你的数据节点使用的是真正的内存而不是交换空间。如果数据节点使用交换空间启动的话,性能会急剧下降。这不仅仅是性能问题,还可能影响集群的稳定性。

## 为数据节点使用处理器亲和度

在多 CPU 机器中,将数据节点进程锁定在与网络通信无关的 CPU 上。在某些平台(如 Sun CMT 处理器系统)上修改配置文件可以达到这个目的,即在 [ndbd] 段使用 LockExecuteThreadToCPU 或 LockMaintThreadsToCPU 参数。

如果遵从上述最佳实践,就可以使 MySQL 集群达到最佳性能和最高可用性。想了解更多关于优化 MySQL 集群的信息,请参见白皮书“MySQL 集群数据库的性能优化”部分。

# 小结

本章讨论了使用 MySQL 集群达到高可用性的方案。MySQL 集群的强大之处在于将表分区,然后将它们分布到独立的节点上去,MySQL 集群的并行架构与多主(multimaster)数据库相当。这样系统就可以并发地执行大量的读写操作。所有的更新对所有的应用程序节点(通过 SQL 命令或 NDB API)立即可用,这些应用程序节点访问存储在数据节点上的数据。

因为写负载被分布到所有数据节点上,所以你可以获得很高的写吞吐量,同时扩展了事务型负载。最后,通过多个 MySQL 服务器节点(即 SQL 节点)并行运行,其中每个服务器通过多个连接共享负载,以及使用 MySQL 复制确保不同地理位置站点之间的数据传输,你就可以构建高性能、高开发的事务型应用程序。

尽管很少有应用程序有这样严格的需求,但 MySQL 集群还是非常适合那些对 MySQL 高可用性有极高要求的应用程序。

“Joel!”

老板折回来站到门口,Joel 笑着问道,“什么事,Bob?”

Summerson 先生走进办公室,关上门,然后拉过一把椅子,坐在 Joel 对面。

顷刻之间 Joel 措手不及,只是笑着说,“我可以为你做点什么吗,Bob?”

329



“你已经帮我做过了，Joel。你很快搞定了 MySQL 的事儿，也跟上了我们加快发展和近期收购的步伐，现在你又帮我在这笔买卖中赚了很多钱。我知道我给你安排了很多任务，你也应该得到一些回报了。”一个令人不安的停顿之后，他问道，“你打高尔夫球吗，Joel？”

Joel 耸了耸肩，“大学以后就不玩了，而且我也向来打不好。”

“没关系。我喜欢打，但它好像不喜欢我，我每次打球都会丢掉一半的球。周六有时间玩 9 个洞吗？”

Joel 不知道哪里不对，但直觉告诉他应该接受这个请求。“当然，我有时间。”

“很好，我们上午 10 点在 Fair Oaks 碰面。我们先玩 9 个洞，然后午饭的时候讨论一下你的将来。”

“好的。到时候见，Bob。”

Summerson 先生站起来，打开门，然后停住了。“我已经让会计处给你做个预算，包括 MySQL 企业版的订购费，和雇佣两个全职助手的费用。”

“谢谢。”Joel 感动地说。他还没有准备好完全接受他的提议，更不用说还有更多的职责。

Summerson 先生消失在大厅后，Joel 的朋友 Amy 走进来站到他旁边，“你还好吧？”她关心地问。

330 ➤ “好啊，怎么了？”

“我以前从没见他关门跟别人谈话的。如果你不介意我多问的话，你们都说什么了？”

Joel 挥了挥手，说：“他叫我去玩高尔夫，然后说我有自己的预算了，可以订购 MySQL 企业版。”

Amy 笑了，抓住他的胳膊，“那就好，Joel，真好。”

Joel 困惑了，他觉得负责管钱或者同意购买订单不至于有这么大反应。“怎么了？”

“上一个和 Summerson 先生打高尔夫的人升职加薪了。Summerson 看起来很无情，其实还蛮值得为他卖命的。”

“真的吗？”Joel 盯着桌上的文件，告诉自己不要期望太高。

“中午一起吃饭吗？”Amy 轻轻挤了一下他的胳膊问道。

Joel 看了看 Amy 放在自己胳膊上的手笑着说，“好啊，我们去个好地方吧。”但在接受了她的要求之后，Joel 知道可能要为了下一次约会而熬夜工作了。



## 监控和管理

现在，我们有了一个复杂的多服务器系统。你希望它能够满足站点的需求，所以你必须控制好它。这一部分将解释监控的概念，以及一些性能方面的话题，并讲述备份和故障处理的其他问题。

# 监控入门

Joel 将脱脂低卡拿铁咖啡、水果杯和奶酪糕点放到桌子上，对着这堆吃的笑了。自从在上班的路上发现了一家高档购物中心后，他的早餐就变得相当有创意了。

他一边打开拿铁的盖子，一边打开显示器等待邮件应用收取消息。他浏览着这些邮件的标题，希望别再收到老板发来的消息。这时，他注意到有几个标题与性能问题有关的新消息。

Joel 点开这些信息，看了一下内容。看到有人抱怨查询数据库系统的应用程序的响应时间太长，他嘴里嘀咕着：“嗯，我想一定是什么地方出问题了。”

他边打开糕点边思索着是什么导致出现这些问题。“昨天还是好好的呢，”他想。喝了几口拿铁后，他想起了在大学实验室工作时读到的一些关于性能监控的知识。

Joel 吃完糕点后拿起《高可用 MySQL》这本书，说：“这本书中肯定有我要的东西。”

怎么知道服务器什么时候性能变差的？等到用户告诉你系统有问题时，这些问题可能已经存在有一段时间了。如果问题长时间得不到解决，将会使系统诊断和修复过程复杂化。

这一章，我们首先讨论使用系统的各种基本工具，在操作系统层面对 MySQL 进行监控。我们从这里开始是因为系统服务或应用程序总是依赖于操作系统及其本身硬件的性能。如果操作系统性能很差，那么它上面安装的数据库系统或应用程序的性能也好不到哪里去。

我们首先考察为什么要使用监控系统，再看看在主流操作系统上的基本监控任务，然后讨论监控是如何简化预防性维护工作的。一旦你掌握了这些技巧，就可以更加了解你的数据库系统。下一章我们将进一步讨论监控 MySQL 服务器的更多细节，并介绍一些解决常见性能问题的实用指南。

## 监控方法

当我们想到“监控”时,通常会联想到一些用于监测问题的早期预警系统。然而,监控(作为动词)是指“在不影响操作或运行环境的前提下,用仪器观察、记录或检测操作或环境。”这种早期预警系统采用自动采样和预报系统相结合的方式进行预警。

Linux 和 UNIX 操作系统非常复杂,有很多参数会影响到系统主要和次要进程的行为。优化这些操作系统的性能可以说是一门艺术而非科技。与一些桌面操作系统不同, Linux 和 UNIX (及其变种)既不隐藏系统优化工具,也不限制你对系统的优化。而一些桌面操作系统,如 Mac OS X 和 Windows,则将系统的许多基础机制隐藏在用户非常友好的可视化界面后面。

例如, Mac OS X 操作系统是一个非常优雅而且运行流畅的操作系统,通常情况下它只需要一点点或根本不需要用户的注意。然而,在稍后的章节你将看到, Mac OS X 操作系统提供了许多先进的监控工具,这些工具可以帮你优化 Mac OS X 操作系统。

Windows 操作系统有很多版本,目前最新的版本是 Windows 8。幸运的是,这些版本中的大部分都包含相同的监控工具,用户可以使用这些监控工具优化系统以满足特定需求。虽然 Windows 不如 Mac OS X 操作系统那样讨好,但是它提供了许多用户可访问的优化选项。

系统监控主要有 3 种类型:系统性能、应用程序性能和安全性。实际监控的原因可能更加具体,但是通常任何监控任务都可以归为这三类中的一类。

每一类监控都使用不同的工具集(有一些重叠),而且监控目标也不相同。例如,监控系统的性能是为了确保系统能够以最高效率运行,监控应用程序的性能是为了确保单个应用程序能够以最高效率执行,而监控安全性是为了确保系统被保护在最安全的状态。

335

监控 MySQL 服务器类似于监控应用程序。这是因为就像大多数数据库系统一样, MySQL 需要衡量许多与操作系统关系不大或无关的变量和状态指标。但是,数据库系统很容易受到主机操作系统的影响,所以在进行数据库系统的问题诊断前,确保主机操作系统运行良好是很重要的。

我们的目的是监控 MySQL 系统以确保数据库系统能够以最高的效率执行,所以接下来将讨论操作系统的性能监控。至于安全性监控,将在其他章节进行详细介绍。

## 监控的好处

监控有两种方法。你希望通过监控来确保一切都没有改变(没有性能下降,也不存在安

全漏洞)，或者确定哪里发生了改变或哪里出错。通过监控确保一切不变称为主动监控（proactive monitoring），而通过监控确定哪里出错称为被动监控（reactive monitoring）。遗憾的是，大多数监控是被动监控。很少有 IT 从业人员有时间或资源进行主动监控。因此有些从业人员只知道被动监控这一种形式。

不过，花时间主动监控系统可以减少大量的被动监控工作。例如，如果用户抱怨系统性能差（这是被动监控的首要原因），如果没有以前的监控结果进行对比，就无法知道系统退化了多少。这种记录监控结果的行为又叫为系统“建立基准”（即在某段时间内，分别监控了系统在低、正常和高负荷时的性能）。如果持续频繁地进行采样，就可以确定在各种负荷情况下系统的典型性能。因此，当用户报告性能问题时，对系统进行采样并与基准进行比较。如果历史数据足够详细，通常一眼就能发现系统的哪一部分发生了改变。

## 监控系统组件

监控性能时应该检查系统的四个基本组件。

### 处理器

检查处理器的使用量及利用率达到了什么样的峰值。

336

### 内存

检查内存的使用量，以及还有多少可用。

### 磁盘

检查磁盘空间的可用量，磁盘空间是如何被占用的，还有哪些需求需要占用磁盘空间，以及磁盘的读取速度（响应时间）。

### 网络

检查网络通信的吞吐量、延迟和错误率。

## 处理器

监控系统的 CPU，确保没有失控进程，而且 CPU 周期在各个运行程序之间是平均分配的。要做到这一点，方法之一是确定正在运行的程序列表，然后确定每个程序所占 CPU 的百分比。另外一个方法是检测系统进程的平均负载。大多数操作系统提供了 CPU 性能的多种视图。





进程是指在 Linux 或 UNIX 操作系统上运行的一个工作单元。一个程序可能同时运行一个或多个进程。多线程应用程序，如 MySQL，通常在操作系统中表现为多个进程。

当 CPU 处在高性能负载下且争用激烈时，系统将运行缓慢，甚至出现死机的情况。在这种情况下必须减少进程的数量，或者减少那些消耗 CPU 时间较多的进程的 CPU 使用。要知道哪些进程消耗更多的 CPU，可以使用 Linux 或 UNIX 系统的 *top* 实用程序、Mac OS X 的 Activity Monitor 或者 Windows 的任务管理器中的“性能”选项卡。但是要保证一定要监控 CPU，确定 CPU 的高使用率确实会产生问题——系统运行缓慢更有可能是内存争用导致的，下一节将讨论这个问题。

CPU 超载的一些常见的解决方法有：

#### 添加新服务器运行某些进程

这当然是最好的办法，但是新服务器需要资金。有经验的系统管理员通常可以找到其他方法减少 CPU 的使用率，尤其是当组织更愿意花时间而不是花钱在这件事情上时。

#### 删除不必要的进程

大量系统为了某些特定场合运行的后台进程，大多数时候会使系统陷入瘫痪。然而，作为一个系统管理员，必须非常了解操作系统，知道哪些进程是不必要的。

#### 杀死失控的进程

这些失控进程可能是有缺陷的应用程序导致的，而且当间歇或偶尔出现性能问题时，它们往往是罪魁祸首。倘若使用某种可控的或有序的方法无法停止失控进程，可能需要使用强制退出对话框或命令行强制性地终止这个进程。

#### 优化应用程序

有时候应用程序占用的 CPU 时间或其他资源比它们实际需要的要多。设计糟糕的 SQL 语句经常会拖累数据库系统。

#### 较低的进程优先级

有些进程可以作为后台作业运行，例如报表生成器，而且它们可以运行得慢一些，以给互动进程腾出空间。

#### 重新安排进程

也许有些报表生成进程可以在夜间运行，那时系统负荷较低。

消耗太多 CPU 时间的进程被称为是 CPU 受限的 (CPU-bound)，即不会为了等待 I/O 而

挂起，也不能被换（swap）出内存。

如果没有CPU争用，而且仅有少数几个进程在运行或者不存在消耗大量CPU时间的进程，那么性能问题可能是其他原因导致的，例如等待磁盘 I/O、内存不足、过度页交换等。

## 内存

监控内存是为了确保应用程序不要请求过多的内存，因为内存过多会浪费很多系统时间用于内存管理。从最初有限的随机存取存储器（即 RAM 或主存），操作系统已经演变到使用磁盘存储器来存储主存中未使用的部分或页。这种技术被称为分页（paging）或交换（swapping），通过存储挂起进程的内存，然后在进程被激活时获取内存，这样系统能够一次性装载比主存容量更多的进程。虽然在内存页与磁盘间来回移动的代价相对较高（与直接访问主存相比，比较耗时），但是现代操作系统在这方面能够做到很快，如果处理器和磁盘的运行速度能够跟上需求，那么这种缺陷就不是问题了。

338 然而操作系统可能会定期执行交换以回收内存。所以一定要测量一段时间的内存使用情况，确保这不是常规清理操作。

在分页操作很频繁的时候，可用内存很少可能是因为失控进程占用了太多内存，或者太多进程请求了太多内存。这种高频率分页被称为 thrashing，处理方式与 CPU 资源争用类似。消耗过多内存的进程被称为是内存受限的（memory-bound）。

在处理内存性能问题时，自然想到的处理方法是增加更多的内存。虽然这样可以解决问题，但内存也有可能在各子系统间分配不均匀。

这种情况下可以做这几件事情。可以为不同系统的组件（如内核或文件系统）或支持内存优化的应用程序（如 MySQL）分配不同数量的内存。还可以更改分页子系统的优先级，让操作系统早点开始分页。



调整服务器的内存子系统时要非常小心。请务必参考特定操作系统的性能提升的相关文档或书籍。

如果在监控内存时发现系统的分页并不频繁，但仍然存在性能问题，这时就可能跟其他的子系统有关。

## 磁盘

监控磁盘使用率是为了确保系统拥有足够的可用磁盘空间和 I/O 带宽，以使进程执行时不会出现明显的延时。使用单进程传输率（per-process）或整体传输率（overall transfer）

衡量读写磁盘的传输速率。单进程传输率是指单个进程读写的数据量。整体传输率是指读写磁盘的最大可用带宽。有些系统有多个磁盘控制器，这时整体传输率是根据每个磁盘控制器衡量的。

如果有一个或一个以上的进程消耗了过多最大磁盘传输率，就会出现性能问题。就像进程消耗太多的 CPU 周期一样，这会对系统其他进程产生不利的影响：会“饿死”其他进程，迫使它们更长时间地等待磁盘访问。

消耗太多磁盘传输率的进程被称为是磁盘受限的（disk-bound），即访问磁盘的频率大于磁盘传输率的可用份额。如果能够减少磁盘受限的进程给 I/O 系统带来的压力，将会为其他进程腾出更多的带宽。

339



在描述进程的时候，你可能还听过术语 I/O 受限（I/O-bound）或 I/O 饥饿（I/O-starved）。通常指的是进程消耗太多磁盘。

满足进程这种大量磁盘 I/O 需求的方法之一是，增加文件系统的块大小，从而使大型传输更有效，减少由磁盘受限的进程带来的系统开销。但是，这可能会使其他进程变得更慢。



如果服务器上只有一个控制器或磁盘，那么在优化文件系统时需要小心。请参考特定操作系统上性能提升的相关文档或书籍。

如果资源充足，那么处理磁盘争用的一个办法是添加磁盘控制器和磁盘阵列，将其中一个磁盘受限的进程的数据挪到新的磁盘控制器上。另一种方法是将磁盘受限的进程移到另外一个使用率较低的服务器上。最后一种方法是，有时候通过将磁盘系统升级得更快，能够增加磁盘的带宽。

至于先从哪里开始优化或者哪种优化方法是最好的，有不同观点。但是我们认为：

- 如果需要运行大量的进程，就需要最大化磁盘传输速率，或者将进程分布在不同的磁盘阵列或系统上。
- 如果只运行少数几个进程但数据访问量大，就需要最大化单进程传输率，通过增加文件系统的块大小来实现。

此外，可能需要在这两种方案之间取得平衡以满足特殊的混合进程需求，即将某些进程转移到其他系统上。



## 网络子系统

监控网络接口以确保系统拥有足够的带宽，而且正在发送或接收的数据具有足够高的质量。

340 有些进程试图读写的数据量超过了网络配置或硬件所允许的范围，以至于它们消耗了过多的网络带宽，这样的进程被称为网络受限的（network-bound）。这类进程为了避免自己发生延时而阻止其他进程访问充足的网络带宽。

网络带宽问题通常由网络接口最大带宽的百分比利用率决定。通过给不同进程分配特定的网络端口，能够解决这个问题。

网络数据质量问题通常表现为网络接口遇到了大量错误。幸运的是，操作系统和数据传输应用程序通常采用校验和（checksumming）或其他算法来检测这类错误，但是重新发送会给网络和操作系统带来沉重负担。解决问题的方法可能有：将某些应用程序转移到同一网络上的其他系统，安装额外的网卡（通常需要网络诊断和更换网络硬件），重新配置网络协议，或者将系统转移到网络上的另一个子网。



当某个进程访问网络子系统的时间太长，我们说它是网络受限的（network-bound）。

## 监控方案

针对以上 4 个子系统，现代操作系统都提供了相应的专用工具，以获取各个子系统的状态信息。这些工具大多是独立的应用程序而且与其他工具不相关（至少无直接联系）。在接下来的章节中你将看到，这些工具都很强大，但是需要大量精力去记录和分析它们产生的所有数据。

幸运的是，有很多第三方监控方案可供大多数操作系统和数据库系统使用。最好让系统提供商推荐能够满足需求并能够与基础设施兼容的最佳监控方案。大多数供应商会提供系统监控工具供选择。下面是一些较出色的监控产品。

up.time (<http://www.uptimesoftware.com/>)

用于监控和报告服务器性能的集成系统，支持多平台。

Cacti (<http://www.cacti.net/>)

RRDtool (<http://oss.oetiker.ch/rrdtool>) 图形数据的图形报表解决方案。RRDtool 是一个开源的日志记录系统，支持 Perl、Python、Ruby、LUA 或 TCL。



KDE System Guard (KSysGuard) (<http://userbase.kde.org/KSysGuard>)

允许用户跟踪和控制进程，配置简单。

Gnome System Monitor (<http://bit.ly/gnome-lib>)

监控 CPU、网络、内存和进程的图形化工具。

Nagios (<http://www.nagios.org/>)

监控所有服务器、网络开关、应用程序和服务的一整套解决方案。

MySQL Enterprise Monitor (<http://bit.ly/ent-monitor>)

为所有 MySQL 数据库的性能和可用性提供实时可见能力。



第 16 章将详细讨论 MySQL Enterprise Monitor 及自动监控和报告。

接下来讨论一些主流操作系统的内置监控工具。我们将更详细地介绍一些 Linux 和 UNIX 命令，因为它们特别适合于研究我们讨论过的性能问题和策略。此外，我们还会介绍 Mac OS X 和微软 Windows 上的监控工具。

## Linux 和 UNIX 监控

Linux 和 UNIX 上的数据库监控工具涉及对 CPU、内存、磁盘、网络甚至安全性和用户的监控。在传统的 UNIX 中，所有的核心工具都从命令行运行，它们大部分位于 *bin* 或 *sbin* 文件夹下。表 10-1 罗列了我们认为有用的工具及其简单描述。

表10-1: Linux和UNIX的系统监控工具

工具	描述
<i>ps</i>	显示系统上运行的进程列表
<i>top</i>	显示根据 CPU 使用率排序的进程活动
<i>vmstat</i>	显示内存、分页、块传输和 CPU 活动的相关信息
<i>uptime</i>	显示系统运行了多长时间。还显示已登录的用户数量以及在 1 分钟、5 分钟和 15 分钟内系统的平均负荷量
<i>free</i>	显示内存使用率
<i>iostat</i>	显示平均磁盘活动和处理器负载情况
<i>sar</i>	显示系统活动报告。允许收集和报告各种系统活动
<i>mpap</i>	显示各个进程分别占用内存的情况
<i>mpstat</i>	显示多处理器系统的 CPU 使用率
<i>netstat</i>	显示网络活动的相关信息
<i>cron</i>	该子系统能够计划进程的执行，从而能够收集某段时间的常规统计信息，或者在特定时间（如在高峰负载或最低负载期间）检查统计信息



有些操作系统还提供了其他可供选择的工具。请参看操作系统文档上有关监控系统性能的其他工具。

从表 10-1 中可以看出，有大量可用的工具能够提供潜在有用的信息。接下来讨论一些比较流行的工具，并简要介绍如何使用它们识别上文提到的那些问题。

## 进程活动

有些命令提供了运行在系统上的进程的相关信息，最重要的是 `top`、`iostat`、`mpstat` 和 `ps`。

### top 命令

`top` 命令提供系统信息的摘要，以及根据 CPU 密集度排序的进程的动态视图。通常，这个命令显示进程的相关信息，包括进程 ID、启动进程的用户、进程优先级、进程的 CPU 占用百分比、进程的耗时，以及启动进程所用的命令。但是，有些操作系统的报告结果略有不同。`top` 命令可能是最常用的实用工具，它每隔几秒就提供一个系统快照。图 10-1 显示了 Linux (Ubuntu) 系统在中等负载情况下运行 `top` 命令得到的输出结果。

```
cbell@cbell-mini: ~
File Edit View Terminal Help
top - 10:15:07 up 1:30, 4 users, load average: 3.80, 2.36, 1.15
Tasks: 153 total, 7 running, 146 sleeping, 0 stopped, 0 zombie
Cpu(s): 25.5%us, 38.9%sy, 0.0%ni, 34.1%id, 0.3%wa, 0.0%hi, 1.1%si, 0.0%st
Mem: 2053520k total, 1131928k used, 921592k free, 279948k buffers
Swap: 2980016k total, 0k used, 2980016k free, 464936k cached

. PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
22479 cbell 20 0 3532 1804 1016 S 13 0.1 0:02.54 bash
26243 cbell 20 0 28344 9524 5324 R 6 0.5 0:00.29 mysqld
7512 cbell 20 0 131m 45m 3352 S 4 2.3 0:09.32 ndbd
7514 cbell 20 0 131m 45m 3300 S 3 2.3 0:08.74 ndbd
11427 cbell 20 0 34232 15m 8340 S 3 0.8 0:12.83 mysqld
11441 cbell 20 0 32632 14m 7936 S 3 0.7 0:09.22 mysqld
7513 cbell 20 0 123m 40m 3364 S 2 2.0 0:08.27 ndbd
3014 root 20 0 294m 23m 9248 S 2 1.1 3:07.58 Xorg
7480 cbell 20 0 17460 3564 2276 S 2 0.2 0:03.09 ndb mgmd
7444 cbell 20 0 13572 9.8m 1316 R 1 0.5 0:03.22 mysql-test-run.
25820 cbell 20 0 2448 1204 912 R 1 0.1 0:00.17 top
7393 cbell 20 0 35476 15m 9732 S 1 0.8 0:16.20 gnome-terminal
7449 cbell 20 0 18108 3916 2276 S 1 0.2 0:03.55 ndb mgmd
719 root 15 -5 0 0 0 S 0 0.0 0:01.33 kjournald
3567 cbell 20 0 45436 19m 13m S 0 1.0 0:37.62 gnome-panel
5955 root 15 -5 0 0 0 S 0 0.0 0:00.44 ksoftirqd/1
25470 cbell 20 0 15972 5804 4740 R 0 0.3 0:00.11 gnome-panel-scr
```

图10-1: `top`命令

系统概要位于列表信息的上方，有一些有趣的数据，包括：用户占用的 CPU 百分比（%us）、系统占用的 CPU 百分比（%sy）、nice 值（%ni，即优先级发生变化的用户进程占用的 CPU 百分比）、I/O 等待的 CPU 百分比（%wa），甚至还有处理软件和硬件中断所占用 CPU 时间的百分比。除此以外，还有可用内存的大小、可用交换空间的大小，多少被占用、多少可用，以及缓冲区的大小等。

系统概要下面是进程列表，这些进程按照占用 CPU 时间的多少降序排列（这就是命令名字的来源）。在这个例子中，第一个任务是 Bash shell，后面跟着几个 MySQL 安装进程。

343

## nice 值

在 Linux 或者 UNIX 系统上能够改变进程的优先级。你可能想要降低这些进程的优先级：它们消耗过多 CPU、不太紧急或者可以稍后运行但你又不想取消或重新安排。这时可以使用 `nice`、`ionice` 和 `renice` 命令改变进程的优先级。

如今大部分 Linux 和 UNIX 都将进程分组，那些改变了优先级的进程被分到 `nice` 组。这样可以获得这些被修改的进程的统计信息，而无须自己记住或整理这些信息。有了这些报告 `nice` 进程的 CPU 占用时间的命令，就能够了解这类进程相对于系统的其他部分所消耗的 CPU 量。例如，如果这个参数值高，表明可能至少有一个进程的优先级过高。

`top` 命令的最佳使用方法也许是让它运行然后每隔 3 秒刷新一次。如果检查这些时间间隔上的显示结果，就会发现哪些进程消耗的 CPU 时间最多，这可以帮你一眼就能确定是否存在失控进程。



可以通过指定命令刷新的时间间隔来改变命令的刷新率。例如，`top -d 3` 将命令刷新的时间间隔设置为 3 秒。

344

大部分 Linux 和 UNIX 的变种系统都有这样的 `top` 命令。有些还有一些有趣的交互热键，能够切换信息是否显示、排序列表，甚至将显示改成彩色的。应当参考特定操作系统的 `top` 命令的使用方法，因为不同的操作系统拥有不同的热键和交互功能。

## iostat 命令

`iostat` 命令可显示几种不同种类的系统信息，包括 CPU 时间的统计信息、设备 I/O 的统计信息，甚至分区和网络文件系统（NFS）的统计信息。这个命令有利于监控进程，因为它提供了系统整体运作的全景图，包括进程和系统等待 I/O 的时间等信息。图 10-2 显示了在中等负载的系统中运行 `iostat` 命令的一个实例。



系统可能默认没有安装 `iostat`、`mpstat` 和 `sar` 命令，但是可以选择性地安装它们。例如，Ubuntu 发行版的 `sysstat` 包中就有这些命令。安装和设置的相关信息详见操作系统文档。



```
cbell@cbell-mini: ~
File Edit View Terminal Help
cbell@cbell-mini:~$ iostat
Linux 2.6.28-15-generic (cbell-mini) 10/13/2009 _i686_ (2 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           10.65    1.09   3.18    2.40    0.00   82.86

Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda                16.69         222.49         366.84    1260455    2078184
sda1               16.68         222.14         366.84    1258473    2078184
sda2                0.00          0.00          0.00         6         0
sda5                0.01          0.29          0.00        1656         0

cbell@cbell-mini:~$
```

图10-2: iostat命令

图 10-2 显示了从系统启动时间起 CPU 使用率的百分比。这些值是由所有处理器的平均值计算而得。你会发现，这个系统运行在双核 CPU 上，却只给出了一行值。这些数据是 CPU 利用率的百分比：

- 345
- 在用户级别执行（运行应用程序）
  - 以 nice 优先级在用户级别执行
  - 在系统级别执行（内核进程）
  - 等待 I/O
  - 等待虚拟进程
  - 空闲时间

这样的报告可以让你了解自启动以来的系统运行状况。虽然可能无法知道系统性能差的时段（因为这是平均值），但它确实提供了一个独特的视角，描述进程是怎样消耗可用的处理时间或等待 I/O 的。例如，如果 %idle 很低，可以确定系统一直很忙碌。同样，高 %iowait 表明磁盘有问题。如果 %system 或 %nice 比 %user 高很多，表明系统失衡，高优先级的进程阻止了普通进程的运行。

## mpstat 命令

mpstat 命令的显示结果与 iostat 命令的处理器时间信息类似，但是 mpstat 将各个处理器的信息分开显示。如果在多处理器系统上运行该命令，将看到单个处理器的数据百分比以及所有处理器的总和。图 10-3 显示了 mpstat 命令的例子。

mpstat 命令有一个参数可以基于间隔时间刷新信息。这个命令有助于查看处理器在一段时间内是如何执行进程的。例如，使用 mpstat 命令可以查看处理器之间是否失衡（即某个特定的处理器被分配了过多的进程）。



```
cbell@cbell-mini: ~  
File Edit View Terminal Help  
cbell@cbell-mini:~$ mpstat  
Linux 2.6.28-15-generic (cbell-mini) 10/13/2009 _i686_ (2 CPU)  
  
10:19:45 AM CPU %usr %nice %sys %iowait %irq %soft %steal %guest  
%idle  
10:19:45 AM all 11.54 1.08 2.99 2.37 0.04 0.14 0.00 0.00  
81.84  
cbell@cbell-mini:~$
```

图10-3: mpstat命令



有些 mpstat 命令实现还提供一个参数查看更多显示结果，包括所有处理器的统计信息。根据具体的操作系统不同，这个参数可能是 -A 或 -P ALL。

346

要了解有关 mpstat 命令的更多知识，请参看操作系统的使用指南。

## ps 命令

ps 命令是我们日常使用的命令之一，但我们从来没有考虑过它的能力和效用。这个命令提供系统上正在运行的进程的快照，显示的信息包括：进程 ID、进程运行的终端、进程已经运行了多长时间和启动该进程的命令。还可以将输出结果传递给 grep，更方便查找进程。例如，命令 `ps -A | grep mysqld` 常常用来查找系统上运行的所有 MySQL 进程。该命令将所有进程的列表传递给 grep 命令，然后仅返回那些含有“mysqld”的行。通过这种方法，确定一个进程 ID，然后使用其他命令获取关于这个进程的详细信息。

ps 命令很灵活，有很多选项可用于显示数据。你可以显示特定用户的进程，也可以通过显示进程树得到某个进程的相关进程，甚至可以改变输出结果的格式。关于特定操作系统的可用选项，请参照相关文档。

使用该命令的输出结果来诊断问题的方法之一是寻找那些已经运行了很长时间的进程，或者检查进程的状态（例如，检查那些处于可疑状态或休眠状态的进程）。除非像 MySQL 这样的已知程序，否则你可能要调查它们为什么运行了这么久。

图 10-4 显示了在中等负荷的系统中运行 ps 命令的一个简短的例子。

这些输出结果的另一个用途是：确定是否存在一些未知进程，或者是否有单个用户运行了大量进程。很多时候，可能是有一个脚本正在大量产生进程，也许是因为这个脚本的配置不合理，甚至可能表明系统存在安全隐患。

347

操作系统还有许多其他内置工具用来显示进程信息。借鉴特定操作系统的性能调优总是进一步了解监控进程的最好来源。

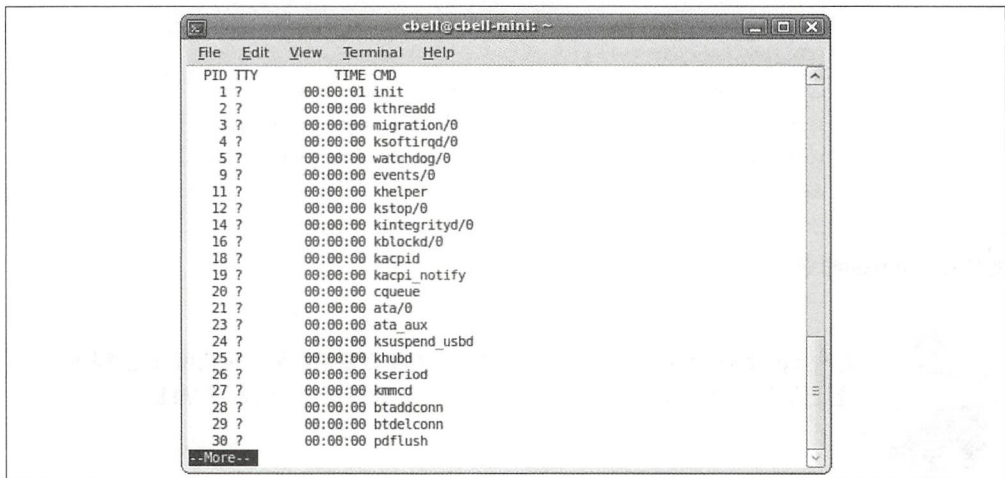


图10-4: ps命令

## 内存利用率

有些命令提供有关系统内存利用率的相关信息。最常用的是 free 和 mpam。

### free 命令

free 命令显示可用的物理内存量，包括总内存量、已用内存量、可用内存量，以及交换空间同样的统计信息，还可显示内核使用的内存缓冲区和缓存的大小。图 10-5 显示了在中等负荷的操作系统上运行 free 命令的例子。

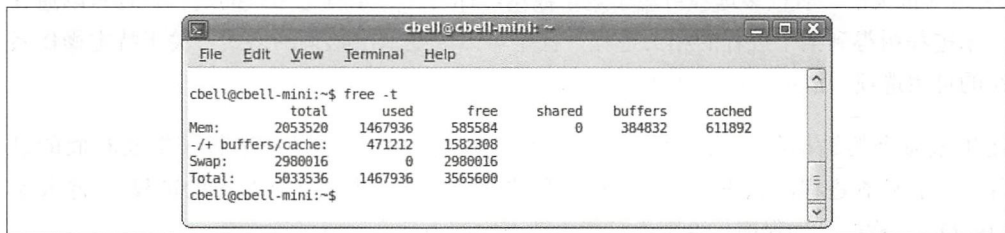


图10-5: free命令

348



图 10-5 是 Ubuntu 系统上 free 命令的输出结果，其中没有 shared 列。

还可以将命令切换成轮询模式，使统计信息按照指定的间隔秒数定期更新。例如，命令

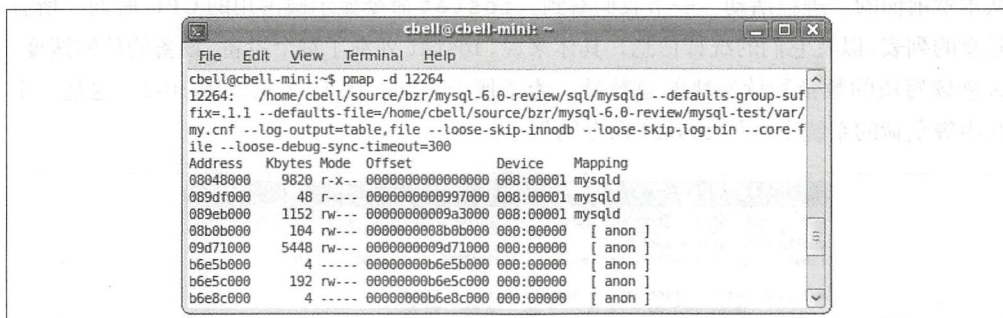
free -t -s 5 表示每隔 5 秒轮询一次内存。

## pmap 命令

pmap 命令提供某个进程内存使用情况的细节映射。要使用此命令，首先必须确定你想查看的进程 ID。可以通过 ps 命令得到进程 ID，或者，如果想确定哪个进程消耗了大量 CPU 时间，可以通过 top 命令获得进程 ID。

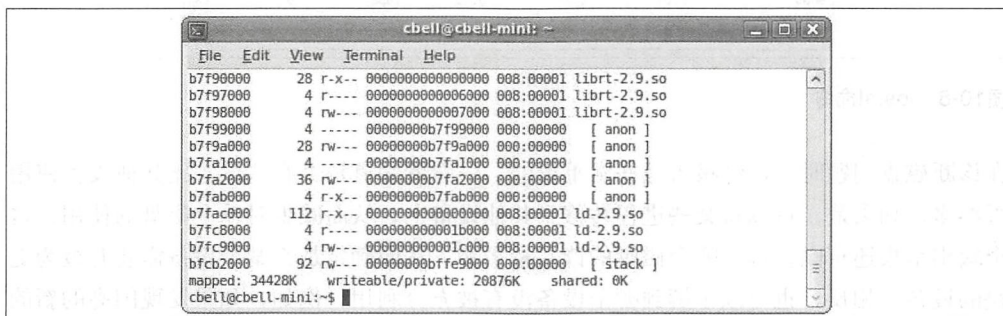
还可以在命令行列举多个进程 ID 来获得多个进程的内存映射。例如，pmap 12578 12579 命令将显示进程 ID 为 12578 和 12579 的进程的内存映射。

pmap 命令的输出结果显示的细节映射包括：所有内存地址的详细信息，以及报告产生时进程使用的内存的大小。它还显示了启动进程的命令，包括完整的路径和参数，有利于确定进程的开始位置和参数。如果想弄清楚为什么进程表现异常，你会惊奇地发现这个命令是多么方便。该命令结果还展示了内存块的模式（即访问权限），这在诊断进程间的问题时非常有用。图 10-6 和图 10-7 显示了在中等负荷的系统上运行 mysqld 进程的一个例子。



```
cbell@cbell-mini: ~  
File Edit View Terminal Help  
cbell@cbell-mini:~$ pmap -d 12264  
12264: /home/cbell/source/bzr/mysql-6.0-review/sql/mysqld --defaults-group-suf  
fix=.1.1 --defaults-file=/home/cbell/source/bzr/mysql-6.0-review/mysql-test/var/  
my.cnf --log-output=table,file --loose-skip-innodb --loose-skip-log-bin --core-f  
ile --loose-debug-sync-timeout=300  
Address Kbytes Mode Offset Device Mapping  
00048000 9820 r-x-- 0000000000000000 008:00001 mysqld  
009df000 48 r---- 0000000000997000 008:00001 mysqld  
009eb000 1152 rw--- 00000000009a3000 008:00001 mysqld  
00b0b000 104 rw--- 0000000000b0b000 008:00000 [ anon ]  
09d71000 5448 rw--- 00000000009d71000 008:00000 [ anon ]  
b6e5b000 4 ---- 00000000b6e5b000 008:00000 [ anon ]  
b6e5c000 192 rw--- 00000000b6e5c000 008:00000 [ anon ]  
b6e8c000 4 ---- 00000000b6e8c000 008:00000 [ anon ]
```

图10-6: pmap命令——第一部分



```
cbell@cbell-mini: ~  
File Edit View Terminal Help  
b7f90000 28 r-x-- 0000000000000000 008:00001 librt-2.9.so  
b7f97000 4 r---- 0000000000006000 008:00001 librt-2.9.so  
b7f98000 4 rw--- 0000000000007000 008:00001 librt-2.9.so  
b7f99000 4 ---- 00000000b7f99000 008:00000 [ anon ]  
b7f9a000 28 rw--- 00000000b7f9a000 008:00000 [ anon ]  
b7fa1000 4 ---- 00000000b7fa1000 008:00000 [ anon ]  
b7fa2000 36 rw--- 00000000b7fa2000 008:00000 [ anon ]  
b7fab000 4 r-x-- 00000000b7fab000 008:00000 [ anon ]  
b7fac000 112 r-x-- 0000000000000000 008:00001 ld-2.9.so  
b7fc0000 4 r---- 000000000001b000 008:00001 ld-2.9.so  
b7fc9000 4 rw--- 000000000001c000 008:00001 ld-2.9.so  
b7fcb000 92 rw--- 00000000b7fcb000 008:00000 [ stack ]  
mapped: 34428K writeable/private: 20876K shared: 0K  
cbell@cbell-mini:~$
```

图10-7: pmap命令——第二部分



349 注意,这里显示的是设备输出格式(在命令行添加 `-d` 参数),以及内存映射或使用的地址。如果需要诊断为什么某个特殊的进程消耗了过多内存以及哪个部分(例如一个库)消耗内存最多,这个命令很有用。

图 10-7 给出了 `pmap` 命令输出的最后一行,显示了一些有用的概要信息。

最后一行显示:有多少内存被映射到文件,私有内存空间的大小,以及与其他进程共享的内存大小。这些信息也许是解决内存分配和共享问题的关键数据。

还有一些其他显示内存利用率的命令和实用工具(例如 `dmesg`,显示开机信息),请参看特定操作系统的性能调优方面的资料。

## 350 磁盘利用率

很多命令可以显示系统的磁盘利用率的统计信息。本节将描述并演示 `iostat` 和 `sar` 命令。

### iostat 命令

从本章前面的“进程活动”一节我们看到, `iostat` 命令显示被占用的 CPU 时间,所有磁盘的列表,以及它们的统计信息。具体来说, `iostat` 列举了每个设备、设备的传输速度、每秒读写块的数量和读写块的总数量。为了便于查阅,图 10-8 重复图 10-2,这是一个在中等负荷的系统上运行 `iostat` 命令的例子。

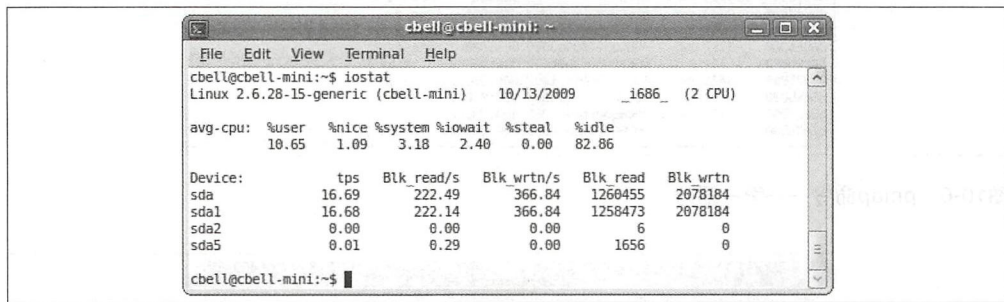


图10-8: iostat命令

在诊断磁盘问题时,这份报告是非常重要的。一看就知道是否有些设备比其他设备使用得多。如果是,可以将某些进程转移到其他设备上,从而减少对单个磁盘的使用。这个输出结果还可以告诉你哪个磁盘的读写量最多,从而确定是否某个设备需要升级为更快的设备。相反,也可以了解到哪个设备没有被充分利用。例如,如果发现闪亮的新的超快速磁盘访问并不频繁,那么可能是你还没有将大量进程配置到新磁盘上,或者程序使用的内存缓存很少有 I/O 操作。



sar 命令

sar 命令是一个非常强大的工具，能够显示有关系统的所有信息。由于它记录了不同时间的数据，能够以各种不同的方式进行配置，所以设置起来有点棘手。请参考操作系统文档，以确保正确地安装它。跟大多数系统命令一样，可以配置 sar，让其定期生成报告。



sar 命令还会显示 CPU 利用率、内存、缓存和类似其他命令的主机信息。有些管理员将 sar 设置为定期运行，提取数据，形成系统的基准。有关 sar 的完整教程已超出了本书的讨论范围。如需更多信息，请参阅由 Gian-Paolo D. Musumeci 和 Mike Loukides 编写的《系统性能优化》(System Performance Tuning) (O'Reilly) 一书。

本节讨论如何使用 sar 命令显示磁盘使用率的信息。为此，我们还显示了 I/O 传输率、交换空间和分页的统计信息，以及块设备的使用率。图 10-9 显示了使用 sar 命令显示磁盘使用情况统计信息的例子。

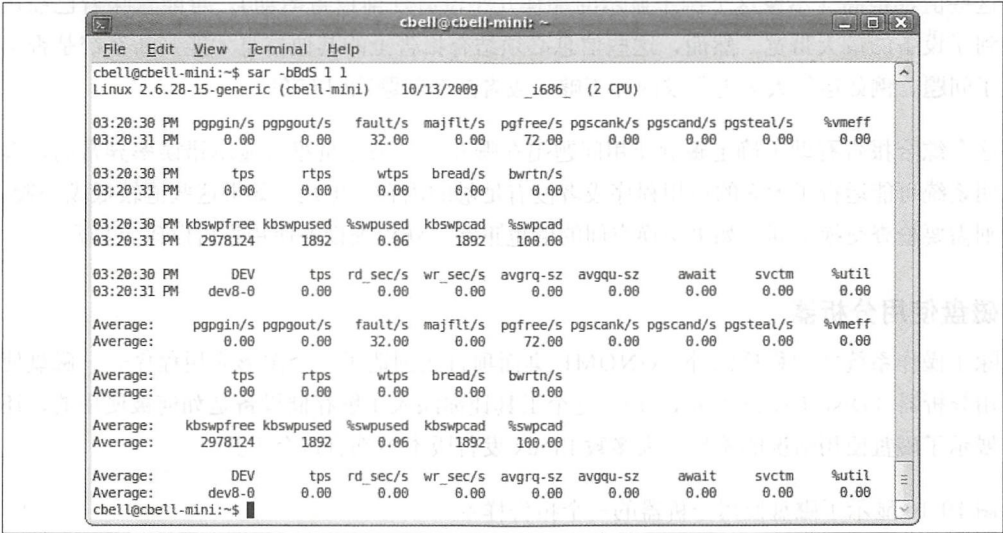


图10-9: 查看磁盘使用率的sar命令

这个报告显示了很多信息，乍一看有些吓人。注意头行后的第一部分，这是关于分页子系统的性能的分页信息。接下来是关于 I/O 传输率的报告，紧接着是交换空间的报告，然后是设备及其统计信息的列表。最后是所有样本参数的平均值。

分页报告显示了分页置换率、每秒内不需要磁盘访问的分页错误数、需要磁盘访问的重大错误数，以及有关分页系统性能的其他统计信息。如果发现大量分页错误（重大分页错误代价更大），可能表明系统运行了过多的进程，这时这个信息就很有用。大量的重

大分页错误会导致磁盘使用问题。也就是说，如果错误数很高而且磁盘使用率很高，那么性能差可能不是磁盘子系统的原因，而是应用程序或操作系统中某些地方出错而导致的。

I/O 传输率报告显示了每秒的事务数量 (tps)、读写请求和读写块的总数量。在这个例子中，系统没有使用 I/O 却具有高 CPU 负载，表明这是一个健康的系统。如果 I/O 值很高，那么可能存在一个或多个进程被卡在了 I/O 受限的状态。对 MySQL 来说，产生的大量随机磁盘访问的查询，或存储在多个磁盘碎片的表，都可能会导致这个问题。

交换空间报告显示了可用交换空间的大小、被使用的交换空间的大小和使用百分比，以及缓存的使用量。这些信息有助于识别换出过多进程带来的问题，而且，跟其他报告一样，还可以帮助确定问题是由磁盘和其他设备导致的，还是由内存或过多的进程导致的。

块设备（系统中以块的方式移动数据的区域，如磁盘、内存等）的报告显示了传输速率 (tps)、每秒的读写速率和平均等待时间。这些信息有助于诊断系统块设备问题。如果这些值都很高（不像这个例子显示的那样几乎没有任何设备活动），可能意味着已经达到了设备的最大带宽。然而，这些信息必须结合报告上的其他信息才能判断系统是否出了问题，例如运行太多进程或内存不够（或者两个问题同时存在）。

这个综合报告有助于确定磁盘使用问题出在哪儿。如果分页报告显示错误率异常高，表明系统可能运行了太多的应用程序或者没有足够的内存。但是，如果这些值较低或一般，则需要检查交换空间；如果交换空间的值也正常，就检查设备使用报告以确定异常。

## 磁盘使用分析器

除了操作系统实用程序以外，GNOME 桌面项目还创造了一个图形应用程序——磁盘使用分析器 (Disk Usage Analyzer)。这个工具让你深入了解存储设备是如何被使用的，还展示了磁盘使用情况的图形。大多数 Linux 发行版本中都有这个工具。

图 10-10 显示了磁盘使用分析器的一个报告样本。

基本上这个报告显示了设备是如何随着分页和交换系统运行的。通常，如果系统中有很多进程换进换出内存，磁盘使用将会出现异常。这就是为什么我们需要在一个报告中同时查看这些项目的原因。

诊断磁盘问题是具有挑战性的，只有几个命令能够显示磁盘使用情况的详细统计资料。但是，有些操作系统提供了更详细、更专业的磁盘使用情况检测工具。不要忘记，还可以通过许多常见命令（如 `ls`、`df` 和 `fdisk`）来确定可用空间大小、什么被挂载、每个磁盘有哪些文件系统等更多信息。参照你的操作系统文档，可以了解所有磁盘相关的命令及其描述，以及磁盘使用率和监控命令。

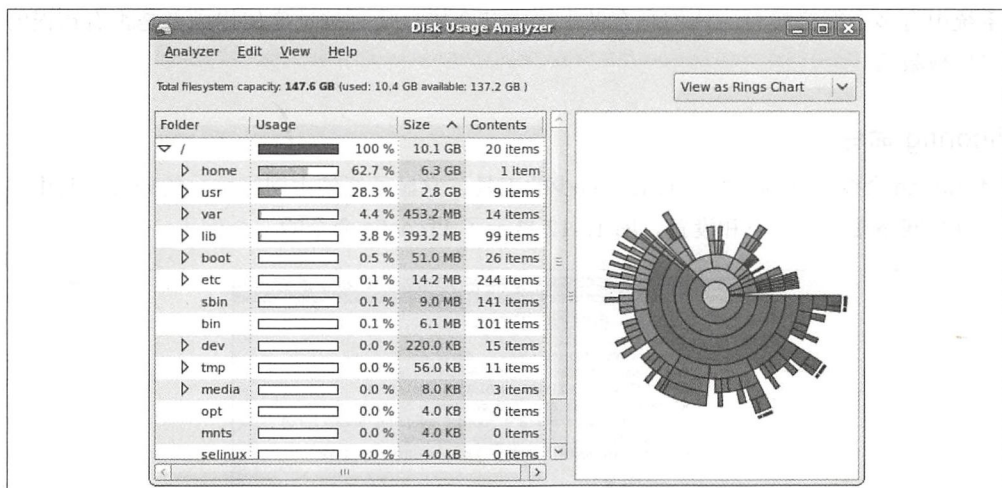


图10-10: 磁盘使用分析器



稍后会提到，`vmstat` 命令也可以显示这些数据，通过 `vmstat -d` 命令可以文本形式显示这些数据。

## 网络活动

网络活动问题的诊断可能需要专业的硬件和网络协议知识。详细的诊断通常交给网络专家解决，但是作为 MySQL 管理员，可以使用 `netstat` 和 `ifconfig` 这两个命令初步了解问题。

### netstat 命令

354

`netstat` 命令可以显示网络连接、路由表、接口统计数据和其他网络相关的信息。网络专家使用这些信息能够诊断和配置复杂的网络问题。不过，它也可以帮助你了解有多少流量正在通过网络接口，以及哪些接口被访问得最多。图 10-11 显示了所有网络接口的样本报告以及每个接口的数据传输量。

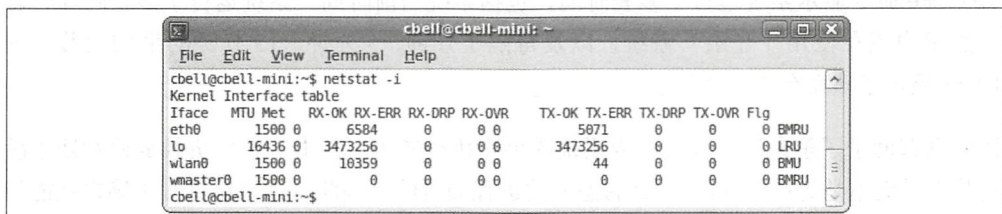


图10-11: netstat命令



系统中有多个网络接口。该信息有助于确定某个接口是否被过度使用或者是否有错误的接口被激活。

## ifconfig 命令

ifconfig 命令是任何网络诊断必不可少的工具，它显示系统中网络接口的列表，其中包括每个网络接口的状态和设置。图 10-12 显示了 ifconfig 命令的一个例子。

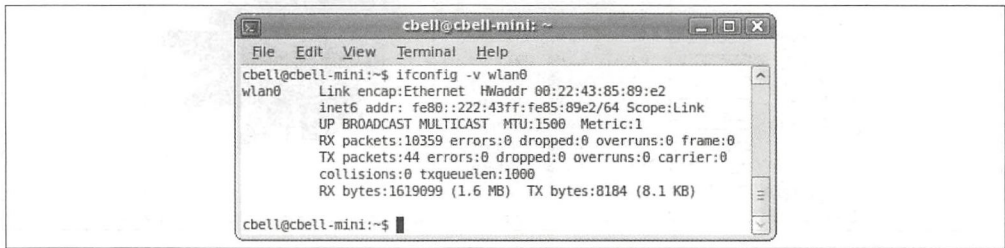


图10-12: ifconfig命令

这里显示了每个接口及其配置信息，无论接口是被激活或被禁止。这些信息有助于确定接口是如何配置的，例如，它可以告诉你网络故障已经转移到一个更慢的接口，而不是通过超高速以太网适配器进行通信。通常，网络问题的根源不是网络流量，而是网络接口的选择和配置。

如果已经有了这份报告，而且还需要诊断网络问题，提前拥有这些报告数据将可以帮助网络专家更快解决问题。一旦减少了消耗太多网络带宽的进程，确定了可用网络接口的位置，网络专家就可以通过配置接口来优化网络性能。

## 常见系统统计信息

我们已经讨论了各个子系统的具体命令，并对这些统计报告命令进行了分组。此外，Linux 和 UNIX 还提供了一些命令获取更多系统相关的常见信息，包括 uptime 和 vmstat。

### uptime 命令

uptime 命令显示系统运行了多长时间，包括系统当前时间、系统运行了多长时间、有多少用户正在使用（登录）系统，以及每隔 1 分钟、5 分钟、15 分钟的平均负载。图 10-13 显示了该命令的一个例子。

该信息有助于了解最近一段时间内系统的平均执行能力。其中，平均负载是针对处于活动状态（而不是等待 I/O 或 CPU 状态）的进程而言的。因此，这个信息对于确定性能问题来说作用有限，但至少能够大致了解系统的健康状况。



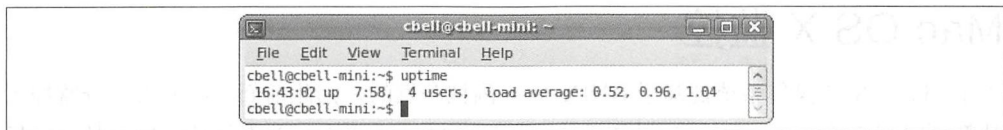


图10-13: uptime命令

## vmstat 命令

vmstat 命令是一个通用的报表工具，提供有关进程、内存、分页系统、I/O 块和 CPU 活动的信息。它有时被用在确定性能问题的第一步。如果某些字段的值过高，可能需要进一步使用本章提到的其他命令去做相关方面的深入检查。

图 10-14 显示了在低负荷系统上运行 vmstat 命令的例子。

这里显示的数据包括进程的数量，其中 r 表示那些等待运行的进程，b 表示那些处于不可中断状态的进程。下一列是交换空间的总量，包括换入 (si) 或换出 (so) 的内存量。然后是接收 (bi) 或发送 (bo) 块的 I/O 报告。接下来是每秒的中断数 (in)、每秒的上下文切换数 (cs)、用户空间上进程运行的时间 (us)、内核空间上进程运行的时间 (sy)、闲置时间 (id) 和等待 I/O 的时间 (wa)。这些时间都是以秒为单位的。

vmstat 命令还有很多参数和选项，详见操作系统手册。

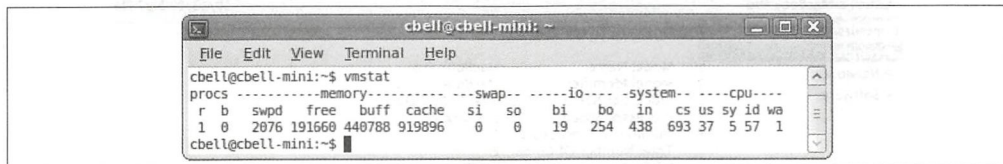


图10-14: vmstat命令

## 使用 cron 自动监控

也许最需要重点考虑的工具是 cron，使用 cron 能够计划进程在特定时间运行。这样就可以运行命令，然后将输出结果保存起来用作后期分析。cron 是一个非常强大的工具，可以获得一段时间内的系统快照。然后，利用这些数据形成系统参数的平均值，当系统日后出现性能不佳时，就可以用这个均值作为基准，与当前系统参数进行比较。从而一眼就能看到哪些发生了变化，为诊断性能问题节省了大量的时间。

如果每天都运行性能监控工具，然后将检测结果与基准进行比较，就可以在用户开始抱怨之前发现问题。事实上，这就是主动监控的基本前提。

# Mac OS X 监控

由于 Mac OS X 操作系统是基于 UNIX Mac 内核开发的, 因此前面介绍的大部分监控工具都可以用来监控 Mac OS X 操作系统。不过, Mac 上也有一些专门的监控工具, 包括以下图形管理工具:

- 357
- System Profiler
  - Console
  - Activity Monitor

本节将逐个介绍以上 Mac OS X 操作系统的监控工具。这些工具构成了 Mac OS X 系统的核心监控和报告工具。在良好的 Mac 设计方案中, 它们被精心编写, 并拥有良好的用户交互界面(GUI)。这些 GUI 甚至能够从文件报告信息。你会发现, 每个工具都非常重要, 都有助于诊断 Mac 系统上的性能问题。

## System Profiler

System Profiler 提供了系统状态的快照。它提供了有关系统的一切细节, 包括所有的硬件、网络和已安装的软件。图 10-15 显示了 System Profiler 的一个实例。

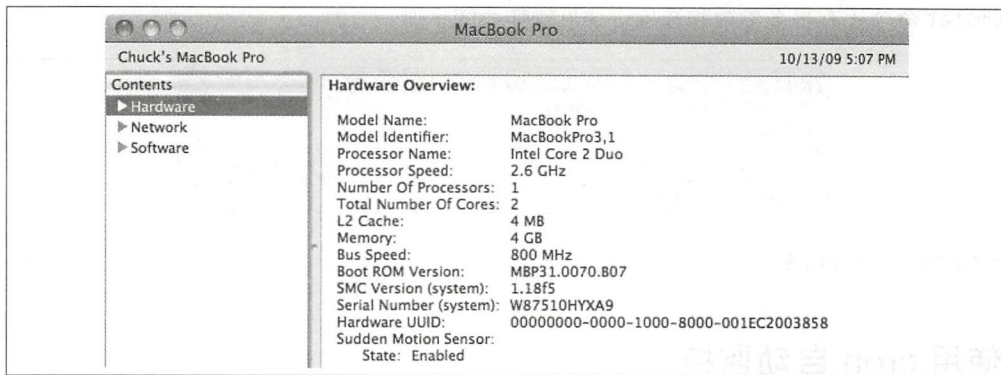


图10-15: System Profiler

在磁盘驱动器的 *Applications/Utilities* 文件夹下找到 System Profiler。或者, 通过 Spotlight 也能够启动 System Profiler。如图 10-15 所示, 左侧是一个树形菜单, 右侧显示详细资料。使用这个树形菜单可以查看系统的各个组件的详情。



如果你更喜欢基于控制台的报告, System Profiler 有一个类似命令行的应用程序, 位于 `/usr/bin/system_profiler`。它有很多参数和选项, 可以限制某些报告的显示。要了解更多信息, 请打开终端, 然后键入 `man system_profiler`。

点开 Hardware 树，将会看到系统上的所有硬件的列表。例如，如果想知道系统上安装的内存类型，可以单击 Hardware 树上的 Memory 条目。

System Profiler 提供了网络报告，我们已经见过 Linux 上另一种形式的网络报告了。单击 Network 树，得到系统上所有网络接口的基本报告。选择树或详细面板上的一个网络接口，可以看到该信息与在 Linux 和 UNIX 上通过网络信息命令产生的信息相同（或者更多）。

另一个非常有用的报告显示了系统上安装的应用程序。单击 Software → Applications，将看到系统上所有软件的列表，包括应用程序名、版本、更新时间、是否是 64 位应用程序以及软件类型（比如，是通用类型还是本地 Intel 二进制类型）。软件类型是非常重要的信息，例如，通用类型比 Intel 二进制类型运行得慢。提前知道这些信息是比较好的，可以确定一些性能相关的期望值。

图 10-16 显示了这个报告的一个实例。

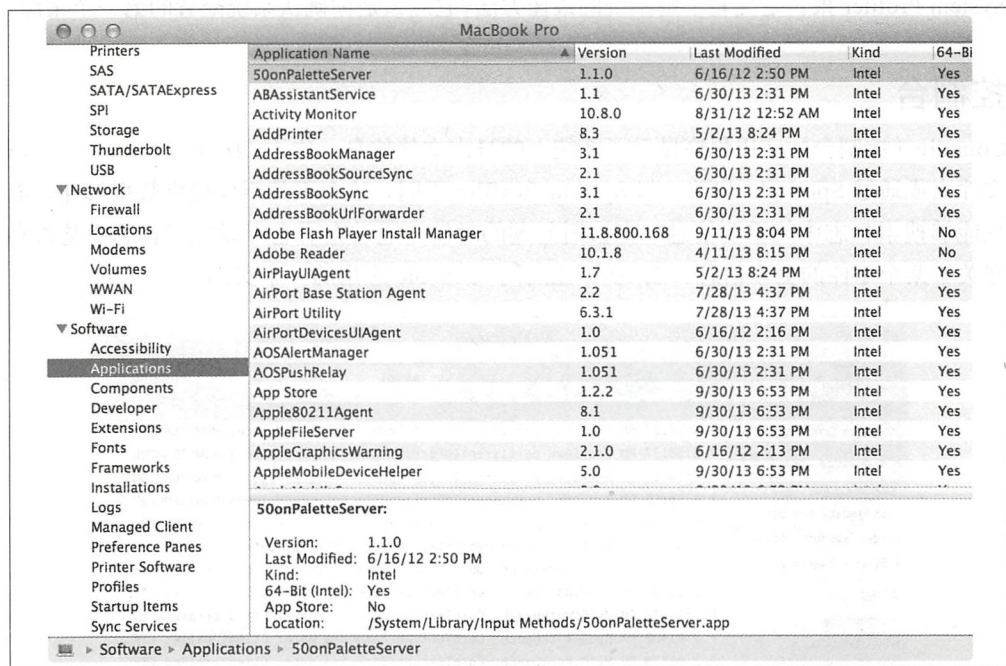


图10-16: 来自System Profiler的内存报告

如你所见，这里有大量的细节信息。可以看到系统安装了多少内存卡和这些内存卡的速度，甚至可以看到它们的制造商代码和部件编号。





我们把每个细节面板都称为一个报告，因为它本质上是某个特定类别的详细报告。有些人将所有数据看成一个报告，这样是不正确的，我们认为将所有东西看成报告的集合会更好。

如果你对这个工具感兴趣，随意单击这个树可获取更多系统相关的信息。从这里几乎可以知道任何事情。

在诊断系统问题的过程中，System Profiler 是非常有价值的。很多时候，AppleCare 专员和训练有素的 Apple 技术人员会要求你提供系统报告。选择 File → Save 命令保存 System Profiler 产生的报告，保存为 Apple 专业人士可以使用的 XML 文件。或者选择 File → Export 命令将报告导出为 RTF。最后把报告保存为 PDF 后打印。

还可以使用 View 菜单更改报告的细节粒度。有 mini、basic 和 full 三个级别，分别从最不详细到最完整提供细节报告。Apple 专业人士通常会要求最完整的报告。

System Profiler 报告是黑箱了解系统的最佳方法，它应该是你确认系统配置的第一个来源。

## 控制台

Console（控制台）应用程序显示了系统上的日志文件信息，位于 `/Applications/Utilities` 文件夹或通过 Spotlight 启动。与 System Profiler 不同，这个工具不仅提供数据转储，而且能够搜索日志中的重要信息。在诊断问题的时候，通过查看它可以确定日志中是否存在关于某个事件的更多信息。图 10-17 显示了 Console 应用的一个实例。

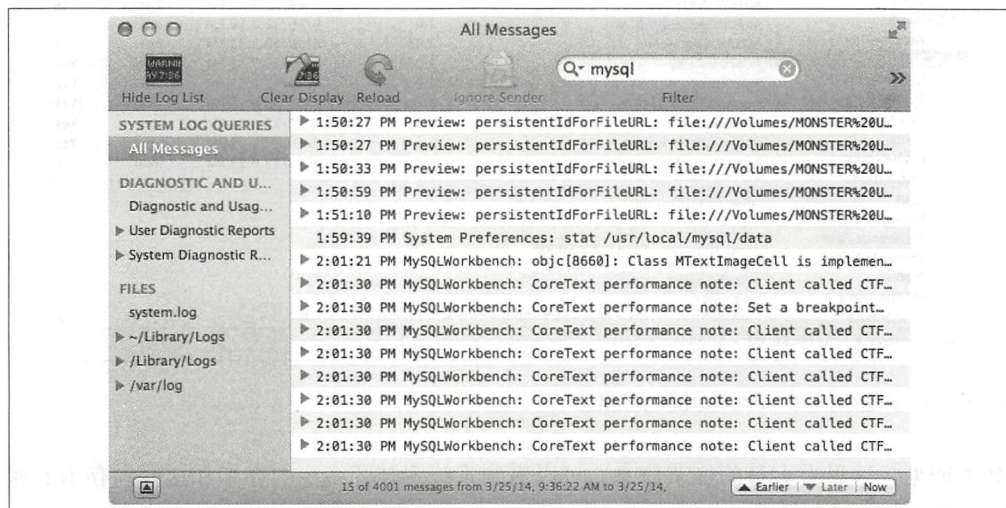


图10-17: Console应用



当启动 Console 应用程序时,它将读取所有的系统日志,并将它们归类为控制台诊断消息。如图 10-17 所示,左边显示日志搜索功能,右边显示日志视图。还可以单击 Files 树上的单个日志文件查看每个日志的内容。日志文件包括以下内容:

#### ~/Library/Logs

存储所有与用户应用程序相关的消息。当出现以下消息时检查这个日志:登录时崩溃的应用程序、磁盘活动信息,以及其他与用户相关的任务。

#### /Library/Logs

存储所有系统信息。当出现以下消息时检查这个日志:系统级崩溃产生的信息和其他异常事件相关的信息。

#### /private/var/log

存储 UNIX BSD 进程相关的信息。当出现以下消息时检查这个日志:系统守护进程或 BSD 实用工具的相关信息。



日志是顺序存储的文本文件,数据总是追加写入,从来不会从中间更新,而且几乎不删除。

Console 最强大的功能是它的搜索能力。给定短语或关键字,可以创建消息报告,以便日后查看。选择菜单中的 File → New Database Search 创建一个新的搜索,然后出现一个通用搜索生成器,通过它来创建查询。创建完成后,可以命名并保存报告以便稍后处理。这是监控棘手的应用程序的一个非常方便的方法。

Console 另一个很好的特性是能够在日志中标记当前时间和日期,从而确定上一次查看日志的时间。如果你有这样的经历,通常我们发现了日志中的几个有趣的信息,但是日后查看的时候却不知道在哪里能找到它们,或者不知道日志检查到哪里了,这时,在日志中做标记就很有帮助。要标记日志,先高亮文件中需要标记的地方,然后单击工具栏中的标记按钮。

虽然报告的数据是启动后日志的静态快照,而且任何报告都局限于此快照,但是也可以为日志中的新信息设置警报。通过 Console → Preferences 打开通知,这些通知通过图标弹出来或者某个延迟过后将 Console 应用调到前台。

Console 应用程序是非常有用的,通过监控发生的事件可以查看系统各个方面的工作情况,还可以查找应用程序或硬件错误。当你面对一个性能问题或其他棘手的事件时,一定要搜索关于应用程序或事件的日志信息。有时候,问题的解决方法就在应用程序产生的消息中。

Activity Monitor 不像先前描述的静态工具，它是一个动态工具，能够提供正在运行的系统的相关信息。可以在 Activity Monitor 中找到解决性能问题所需要的大量数据。事实上，在查看 Activity Monitor 时，会看到与 Linux 和 UNIX 那节提到的所有工具类似的信息，包括 CPU、系统内存、磁盘活动、磁盘使用率和网络接口。

例如，通过 Activity Monitor 可以知道哪些进程正在运行，它们消耗了多少内存，以及每个进程消耗的 CPU 时间百分比。在这种情况下，它的作用与 Linux 中的 top 命令类似。

CPU 面板显示了一些有用的信息，如在用户空间（用户时间）运行所花费的时间的百分比，在系统空间（系统时间）运行的百分比，以及空闲时间的百分比。此外，还显示了线程数和正在运行的进程数，以及一张彩色图显示用户和系统时间。加之上面那个类似 top 的显示面板，这个工具有助于审查与 CPU 受限的进程相关的问题。

图 10-18 显示了 Activity Monitor 显示的 CPU 面板。

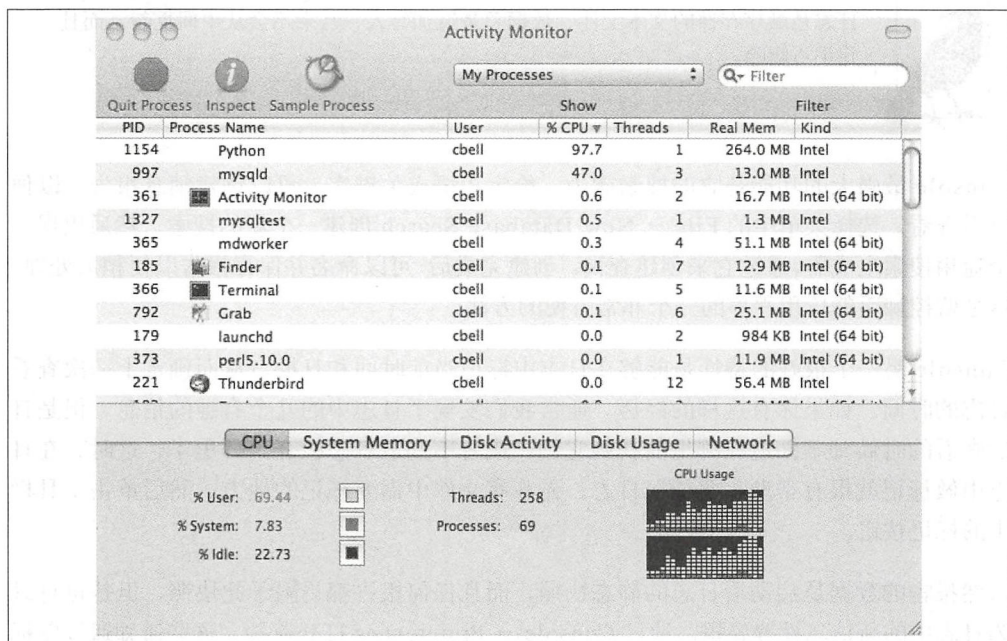


图10-18: Activity Monitor的CPU面板

注意，在这个采样时间里有个 Python 脚本，消耗了相当大的 CPU 时间，这时系统正在终端窗口运行 Bazaar 分支。Activity Monitor 表明了为什么系统在展开代码树时变得缓慢。

双击某个进程可以获得更多有关该进程的信息。还可以通过某种可控的方式取消进程或

强制退出。图 10-19 显示了进程监测对话框的实例。

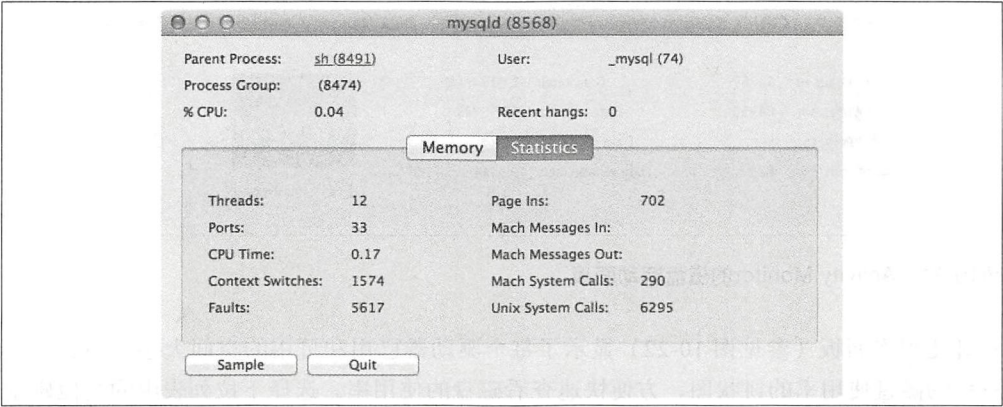


图10-19: Activity Monitor的进程监测对话框



选择 File → Save 可以导出进程列表，可以保存为文本或 XML 文件。在诊断问题时，除了需要 System Profiler 报告外，有些 Apple 专业人士可能还需要进程列表文件。

系统内存面板（参见图 10-20）显示内存分布的信息，包括多少内存是空闲的（free）、多少内存不能被缓存而必须留在 RAM（即联动内存，wired memory）、多少内存被占用（used），以及多少内存是未使用的（inactive）。通过这个报告，可以一眼看出系统是否存在内存问题。 363

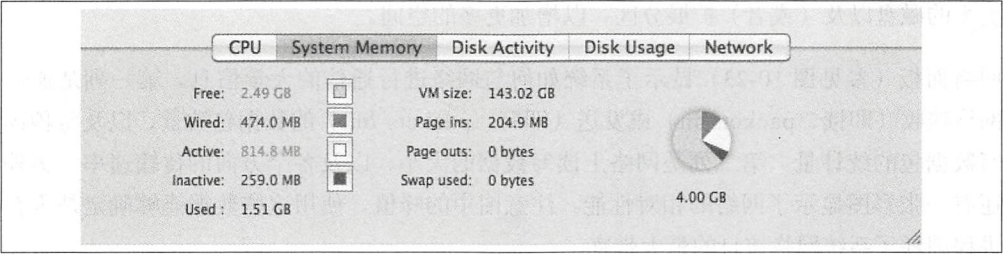


图10-20: Activity Monitor的系统内存面板

磁盘活动面板（参见图 10-21）显示了所有磁盘的磁盘活动。第一列是出（读）入（写）磁盘的数据传输总量，以及磁盘每秒的读写性能。第二列是读写磁盘的数据总大小，以及每个磁盘的吞吐量。还有一个彩色图展示一段时间内的读取情况。

磁盘活动数据可以显示系统是否调用了很多磁盘访问，以及系统读写量（数据总量）是否异常高。如果值异常高，表明可能需要在不同时间运行进程以避免争用磁盘，或者可 364



能需要添加磁盘以平衡负载。

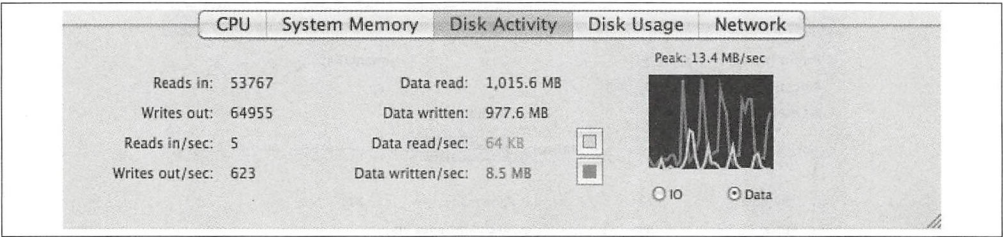


图10-21: Activity Monitor的磁盘活动面板

磁盘使用率面板（参见图 10-22）显示了每个驱动器已用和可用的空间大小，还有一个彩色的磁盘使用率的饼状图，方便快速查看磁盘的使用率。选择下拉列表中的其他磁盘查看其使用率。



图10-22: Activity Monitor的磁盘使用率面板

这个面板可以监控磁盘上的可用空间，从而当系统运行缓慢时，能够知道何时需要添加更多的磁盘以及（或者）扩展分区，以增加更多的空间。

网络面板（参见图 10-23）显示了系统如何与网络进行通信的大量信息。第一列是通过网络接收（即读，packets in）或发送（即写，packets out）的数据包数量，以及每秒读写数据包的统计量。第二列是网络上读写数据的大小，以及各个方向的传输速率。另外还有一张彩图显示了网络的相对性能。注意图中的峰值。使用这些数据能够确定是否有进程消耗了系统网络接口的最大带宽。

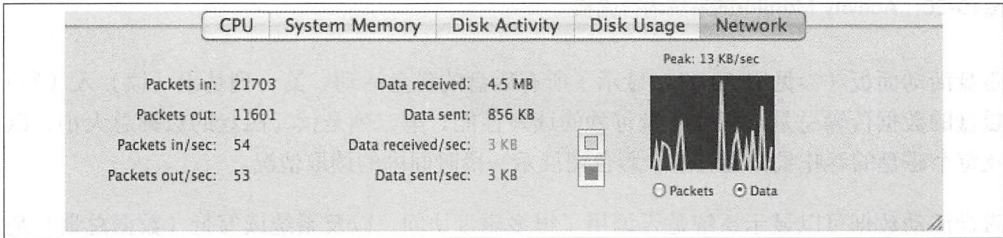


图10-23: Activity Monitor的网络面板



本节简单介绍了 Mac OS X 上的强大监控工具。虽然这不是一个完整的教程，但有助于入门监控 Mac OS X 系统。想要了解每个应用的完整细节，请参看每个应用程序的 Help 菜单提供的文档。

## Microsoft Windows 监控

Windows 背负着缺乏工具的名声，有人说它的监控是有悖常理的。好消息是监控 Windows 系统的障碍是一个错误的认识。事实上，Windows 附带了一些功能强大的工具，包括运行任务的调度工具，能够进行性能快照、在事件查看器（相当于 Windows 的日志）中检查错误，以及实时监控系统性能。



这一小节的图片是从几台 Windows 机器上截取下来的。这些工具与 Windows XP 或更新的版本（包括 Windows Server 2008 和 Windows 8）差别不大。然而，在 Windows 7 或更新的版本中，这些工具的使用方式有所不同，而且每个工具也有区别。

事实上，Windows 管理员可以使用很多工具。这里我们不会介绍所有工具，而是重点介绍实时监控 Windows 系统的工具。首先我们看一下基础报表工具。

以下是可以用于诊断和监控 Windows 性能问题的最流行的工具：

- Windows 体验指数（Windows Experience Index）
- 系统健康报告（System health Report）
- 事件查看器（Event Viewer）
- 任务管理器（Task Manager）
- 可靠性监视器（Reliability Monitor）
- 性能监视器（Performance Monitor）

关于微软 Windows 性能、工具、技术和文档的信息的完美来源是微软 Technet 网站（<http://bit.ly/win-client>）。

## Windows 体验

如果想快速了解系统相对于微软硬件性能指标的执行情况，可以运行 Windows 体验报告。

要启动 Windows 体验报告，先单击“开始”按钮，选择“控制面板”→“系统和维护”，然后择“性能信息和工具”。需要确认用户账户控制（User Account Control (UAC)）才能继续后面的操作。

或者，通过“开始”菜单的搜索功能来访问系统健康报告。单击“开始”按钮，在搜索

框中输入“性能”，然后单击链接“性能信息和工具”。单击“高级工具”，然后选择对话框底部的“生成系统健康报告”链接。需要确认 UAC 才能继续后面的操作。



微软已经改变了 Windows 7 中的 Windows 体验。该报告与之前的 Windows 版本非常相似，但是它提供了更多判断系统性能的信息。

这个报告在安装后只运行一次，但是可以通过单击“重新运行评估”重新生成报告。

这个报告评估了系统性能的 5 个方面：处理器（CPU）、内存、视频控制器（图形）、视频图形加速器（游戏图形）和主硬盘。图 10-24 显示了 Windows 体验报告的范例。

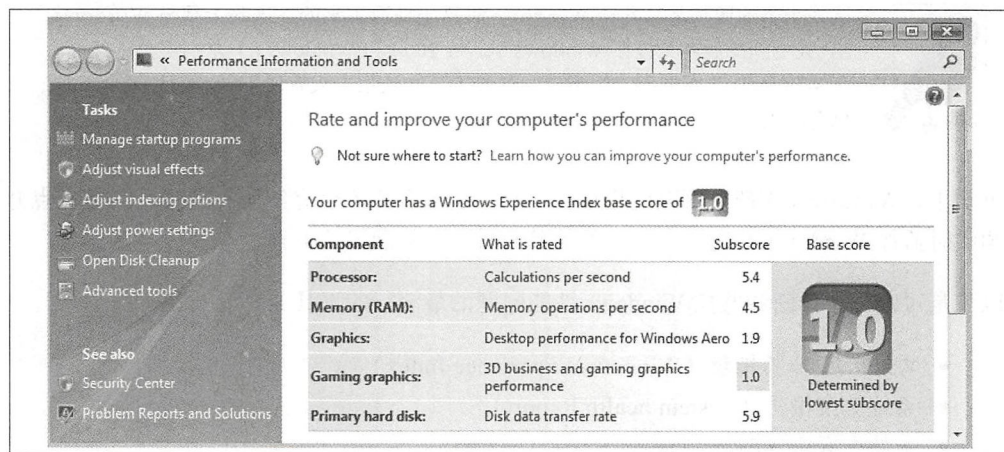


图10-24: Windows体验报告

这个报告还有一个鲜为人知的有用功能。单击“提高计算机性能的提示”链接，可以获得提高这些评分的最佳实践。



每次修改系统配置都应该运行该报告，重新生成指标。这将帮助你识别哪些配置变更影响了系统性能。

367 该工具的最大用处是：在无须分析大量指标的情况下获取系统运行情况的整体印象。任何类别的低分值都可能表示一个性能问题。例如，检查图 10-24 中所示的报告，你会发现该系统有非常低的“图形”和“游戏图形”分值。如果 Windows 系统作为虚拟机或无头服务器运行的话，这是无所谓的，但是会让那些刚刚为高端游戏系统掏出几千美元的人感到不快。

# 系统健康报告

Windows Vista 及其后面的版本有一个独特的诊断改进功能：能够产生系统中所有硬件、软件和性能指标的快照的报告。这与 Mac OS X 的 System Profiler 有些类似，此外还包含性能计数器。

要启动系统健康报告，首先单击“开始”按钮，然后选择“控制面板”→“系统和维护”→“性能信息和工具”。下一步，选择“高级工具”，然后单击对话框底部的“生成系统健康报告”链接。需要确认 UAC 才能继续后面的操作。

或者，通过“开始”菜单中的搜索功能来访问系统健康报告。单击“开始”按钮，然后在搜索框中输入“性能”，选择“性能信息和工具”。单击“高级工具”，然后选择对话框底部的“生成系统健康报告”链接。另一种访问系统健康报告的方法是使用“开始”菜单中的搜索功能。单击“开始”按钮，然后在搜索框中输入“系统健康报告”，然后单击相关链接。需要确认 UAC 才能继续后面的操作。图 10-25 显示了系统健康报告的一个实例。

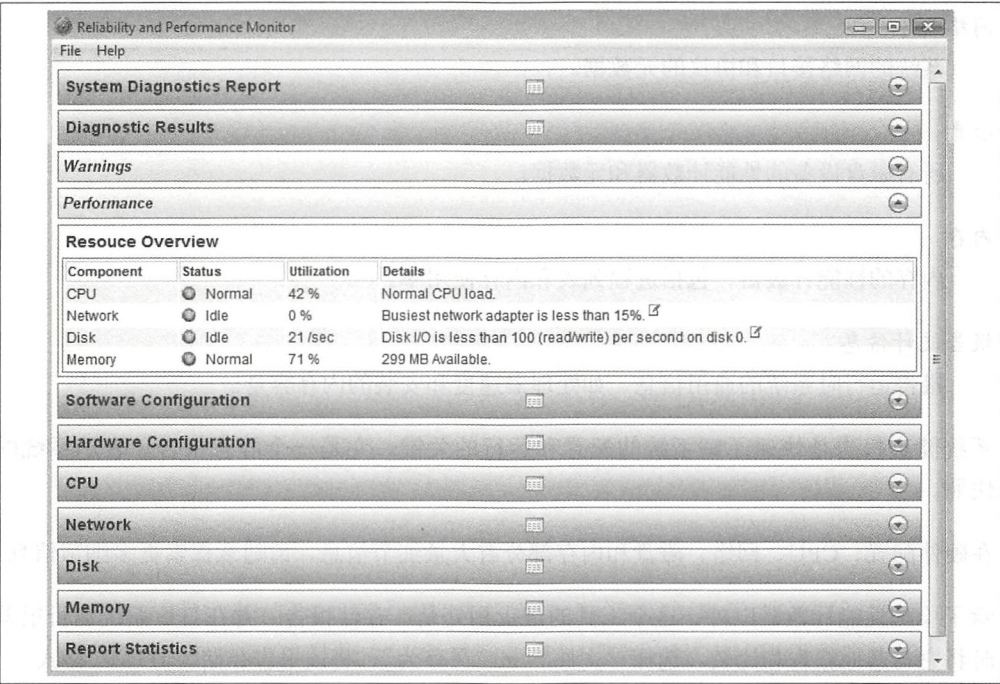


图10-25: 系统健康报告

这个报告包含了所有信息——所有硬件、软件和系统的其他方面都记录在该报告中。注意，这个报告分为几个部分，可以展开或折叠报告以方便查看。下面的列表简单介绍了



每个部分所显示的信息。

#### 系统诊断报告

系统的名称和报告生成的日期。

#### 诊断结果

报告运行过程中产生的警告信息，识别计算机上的潜在问题区域。此外，还有报告运行时的资源概述。

#### 软件配置

系统上安装的所有软件的清单，包括系统安全设置、系统服务和启动程序。

#### 硬件配置

关于磁盘、CPU 性能计数器、BIOS 信息和设备的重要元数据列表。

#### 369 CPU

报告时期运行的进程列表，以及有关系统组件和服务的元数据列表。

#### 网络

系统的网络接口和协议的元数据。

#### 磁盘

所有磁盘设备的性能计数器和元数据。

#### 内存

内存的性能计数器，包括进程列表和内存使用率。

#### 报告统计信息

报告运行时系统的通用信息，如处理器速度和安装的内存容量。

系统健康报告是快速了解系统的配置和运行的关键。它是一个静态报告，表示系统的快照。

在硬件配置、CPU、网络、磁盘和内存部分有大量细节信息，请随意探索更多细节信息。

除了检测性能计数器以外，这个工具的最大用处是：存储报告，并在日后系统运行很差时将与其他报告相比较。选择“文件”→“另存为”，将该报告存储为 HTML 格式。

可以将这个保存的报告作为系统性能的基准。如果依次在低、中和高使用率的情况下生成了多个系统报告，可以将它们放在一起形成系统性能的一般期望。这些期望很重要，可以通过它们确认系统性能问题是否在期望范围内。如果在某个应该是低负载的时间内



系统负荷异常高，用户就会抱怨。如果有这些报告进行比对，则可以节约大量调查系统变慢的确切根源的时间。

## 事件查看器

Windows 事件查看器记录了应用程序、安全性和系统事件的所有消息。这是已发生的（或还在发生的）事件的重要信息来源，应该是诊断和监控系统的主要工具之一。

使用事件查看器可以完成很多事情。例如，生成任何日志的个性化视图，保存日志以便日后诊断，以及为将来的特定事件设置警报。这里我们主要查看日志。如需了解有关事件查看器的更多信息，以及怎样设置个性化报告和事件订阅，请参看 Windows 帮助文档。

370

要启动事件查看器，单击“开始”按钮，然后右键单击“计算机”并选择“管理”项。必须确认 UAC 才能继续后面的操作。然后单击左侧面板中的“事件查看器”。还可以通过以下方法启动事件查看器：单击“开始”按钮，在搜索框中输入“事件查看器”，然后按回车键。

该对话框有三个默认的窗口。左边面板是一个树形视图，包括：自定义视图、Windows 日志、应用程序和服务日志。日志显示在中间面板，而右边面板显示“操作”菜单项。日志条目默认按日期和时间降序排列，所以首先看到最近的消息。



可以自定义事件查看器视图，甚至可以单击日志列对日志进行分组和排序。

打开 Windows 日志的树形菜单，查看应用程序、安全和系统（包括其他部分）的基本日志文件。图 10-26 显示了事件查看器中展开的日志树。

从日志中可以查看和搜索：

### 应用程序

用户应用程序和操作系统服务生成的所有消息。在诊断应用程序的问题时，查看这里是个不错的选择。

### 安全

记录访问和权限的消息，以及访问任何安全对象的失败尝试。这是与用户名和密码问题相关的应用程序查错的好地方。

### 安装

有关程序安装的消息。这是查找关于安装或删除软件错误信息的好地方。

系统

有关设备驱动和 Windows 组件的消息。这是诊断整个系统或设备问题最有用的一组日志，包括在系统级别运行的设备的所有行为信息。

转发事件

从其他计算机转发来的消息。参看 Windows 文档中关于远程事件日志记录的详情。

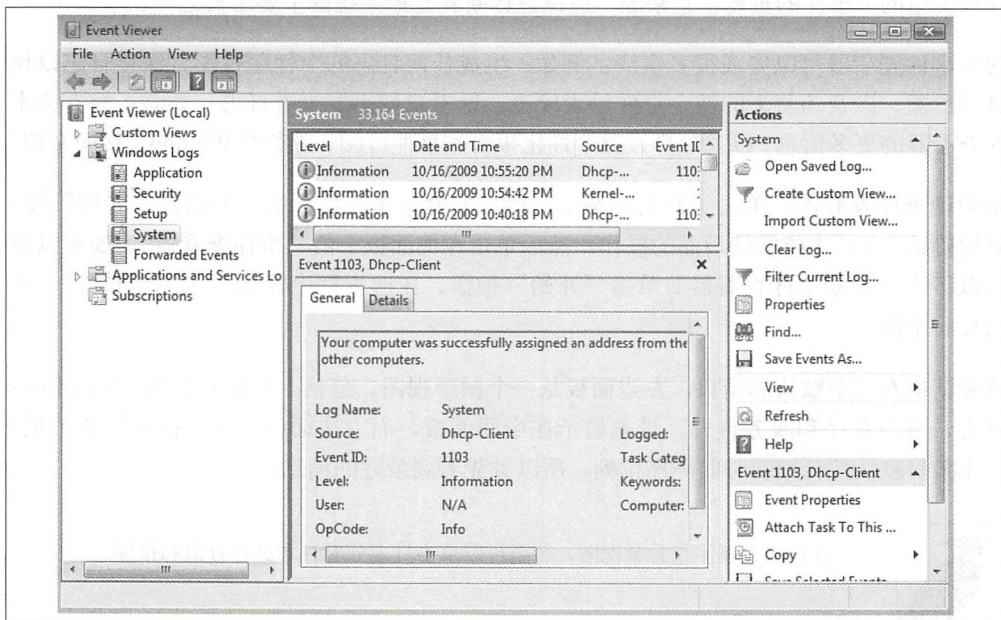


图10-26: Windows事件查看器

371 深究这些日志可能很困难，因为这些信息大部分是开发者所感兴趣的，而对于一般人来说可读性并不强。为方便起见，可以通过以下方法搜索任何日志：单击“操作”面板中的“查找”操作，并输入一串字符。比如，如果你对内存问题感兴趣，可以输入“内存”，一组包含“内存”的日志条目将显示在中间面板。



还可以单击“详细信息”选项卡查看细节以简化管理。

每个日志信息都可以归入以下三个类别（适用于用户进程、系统组件和应用程序）：

错误

表示一些重大错误，如失败的进程、内存溢出问题或系统故障。

表示不太严重的情况或需要注意的事件，如低内存或磁盘空间较少。

#### 信息

传达关于事件的数据。这通常不是什么问题，但在诊断问题时可以提供额外信息，如移除 USB 驱动的时候。

打开左边面板中相对应的树查看日志。单击某个消息查看其细节，该消息将显示在日志条目的下面，如图 10-26 所示。在中间面板的下半部分，单击“常规”选项卡查看消息的常规信息，例如记录的语句，什么时候发生的，包含哪些日志信息，以及运行该进程或应用的用户。单击“详细信息”选项卡可查看记录数据的报告，可以选择友好视图或 XML 视图查看这些信息。还可以存储这些信息以便日后查看，XML 视图可以方便地将报告导出到那些能识别该格式的工具。

## 可靠性监视器

Windows 中最有趣的监控工具是可靠性监视器。这个专用工具能够将某段时间内发生的错误事件和重大性能事件绘制在一张图上。

纵轴表示每天，横轴表示当天的性能指标的总和。如果有错误或其他重大事件发生，就会在图上看到红叉。图表的下方是一个下拉列表，包含软件的安装和删除信息、任何应用程序错误、硬件错误、Windows 错误和其他任何错误。

这个工具非常有助于检测系统在一段时间内的性能，能够帮助诊断以下情况：某个应用程序或系统服务过去一直运行正常但开始变得缓慢，或系统开始产生错误信息。这个工具还可以找到事件第一次出现的日期，并告诉你当它运行正常时系统的运行状况。

该工具的另外一个用途是提供随时间变化的系统性能的日基准，有助于诊断设备驱动变更的相关问题（Windows 管理的灾难之一），这种变动可能只有等到系统明显变慢才会被发现。

总之，可靠性监视器让你有机会回头看看你的系统是如何运行的。最棒的是它不需要手工开启，而是自动执行，从日志中提取大量数据，从而自动了解你的系统的历史。



Windows 的重大问题根源之一是硬件的连接和配置。这里我们不讨论这个问题，否则可能要写成一本书才能谈完。好消息是互联网上有大量关于 Windows 的信息，试试自己搜索一下你的驱动器或硬件，或者访问微软支持中心。还有一个很棒的 Windows 工具叫 Sysinternals (<http://bit.ly/winternals>)。



可通过以下方式访问可靠性监视器：单击“开始”按钮，在搜索框中输入“可靠性”，然后按回车键，或者单击“可靠性和性能监视器”。需要确认 UAC。然后单击左侧树形面板中的“可靠性监视器”项即可。图 10-27 显示了可靠性监视器的一个实例。

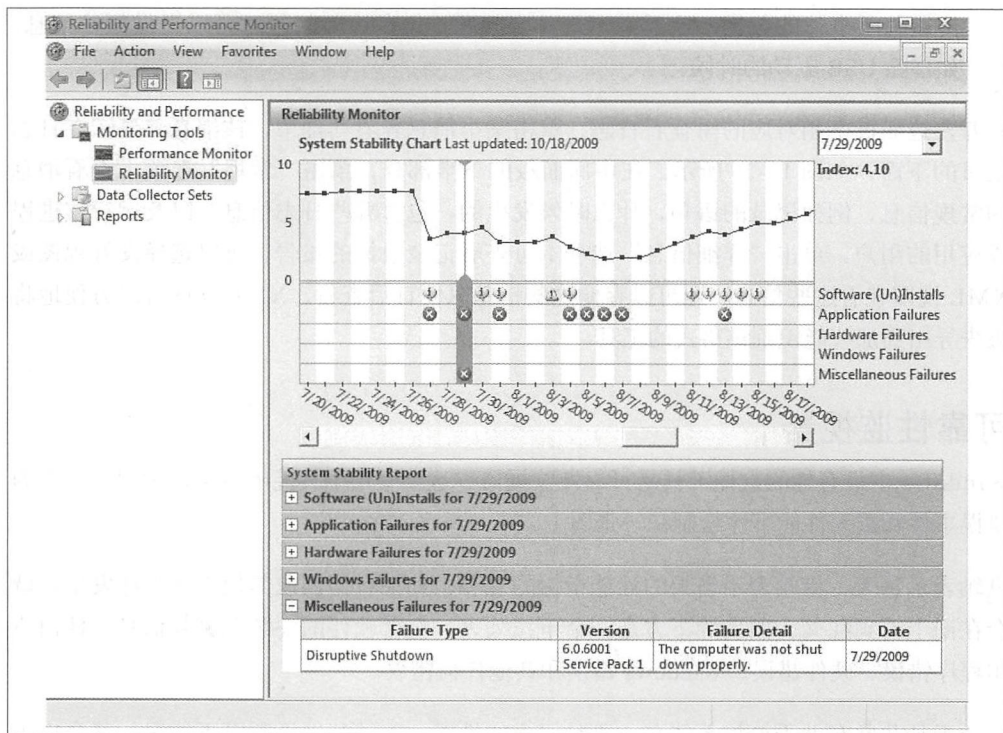


图10-27：可靠性监视器

在 Windows 7 中，可通过以下方式启动可靠性监视器：单击“开始”按钮，紧接着在搜索框中输入“操作中心”，然后按回车键，选择“维护”→“查看可靠性报告”。这个报告与 Windows 以前的版本不同，以更加整洁的方式展示了同样的信息。例如，新的可靠性监视器报告将已知事件的列表以单个列表的形式显示，而不是以下拉菜单的形式显示。

## 374 任务管理器

Windows 任务管理器（参见图 10-28）显示了正在运行的进程的动态列表。这个工具早就有了，而且在不同的 Windows 版本中逐步完善。

任务管理器以选项卡的形式显示了正在运行的应用程序、进程（类似于 Linux 的 top 命令）、系统上启动的服务、CPU 性能、网络性能和用户列表。与其他报告不同，这个工具动态产生数据并定期刷新。因此，这个工具更加适合在低性能时观察系统。



这个报告与系统健康报告显示相同的信息，但是该报告的显示更加紧凑而且信息在不断更新。这里有各种关键指标可用于诊断 CPU、独占资源的进程、内存和网络的性能问题。但这个报告没有磁盘性能报告。

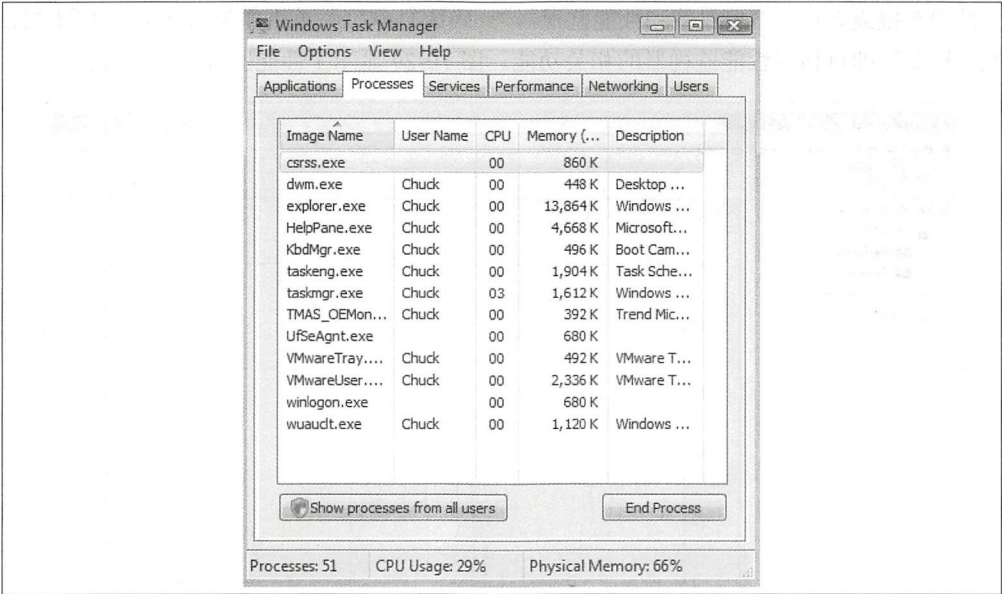


图10-28：任务管理器

任务管理器的一个有趣的功能是：在“开始”菜单栏的通知区域里最小化显示性能计量，以便监视系统的使用峰值。使用组合键 Ctrl + Alt + Del 即可启动任务管理器。

375



运行态的性能监控工具会消耗系统资源，并影响本来性能就很差的系统。

## 性能监视器

性能监视器是跟踪 Windows 系统性能的首选工具，可以选择关键指标，并描绘它们的值随时间变化的曲线。还可以保存当前会话以便日后查看，为系统建立基准。

性能监视器差不多包含了系统所有指标的信息。它有许多关于性能的基本领域（即 CPU、内存、磁盘和网络）的细节信息的计数器，还包含很多其他类别的信息。

要启动性能监视器：首先单击“开始”按钮，选择“控制面板”→“系统和维护”→“性能信息和工具”。然后单击“高级工具”，单击对话框中间的“打开可靠性和性能监视器”。

需要确认 UAC 后才能继续后面的操作。然后单击左侧树形菜单中的“性能监视器”即可访问性能监视器的相关功能。

也可以通过以下方法启动性能监视器：单击“开始”按钮，在搜索框中输入“可靠性”并按回车键或者单击“可靠性和性能监视器”。确认 UAC 后，单击左侧树形菜单中的“性能监视器”即可访问性能监视器的相关功能。图 10-29 显示了性能监视器的示例。

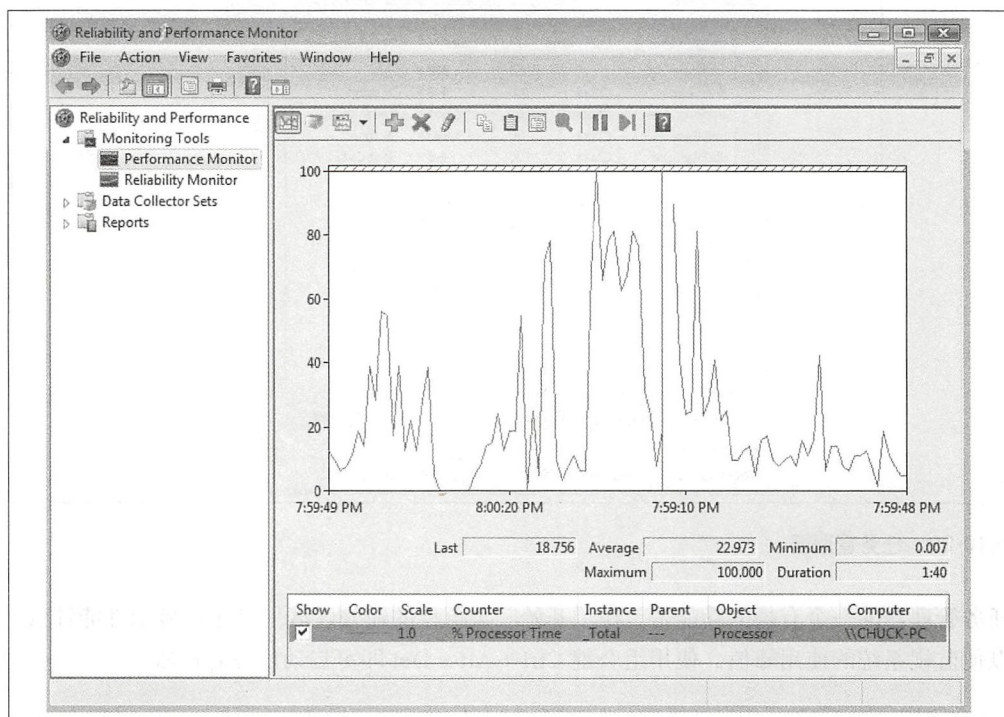


图10-29：性能监视器

微软有两个级别的指标：一个是对象，提供了某个领域（如处理器或内存）的高级视图；另一个是计数器，显示了系统的具体细节信息。因此，既可以监控 CPU 的整体性能，也可以查看更加细节的信息，如空闲时间百分比或正在运行的用户进程数量。单击工具栏上的绿色加号将这些对象或计数器添加到主图上。该操作将会弹出一个对话框，让你从一个长清单中选择需要添加到图中的项。添加项是个简单的过程：先选择对象，然后展开左侧的下拉列表，将对象拖曳到右侧的列表中。

376 想添加多少项都可以，图表的轴会相应地做出变动。如果添加的项过多，或者值差别太大，这个图表将变得不可靠。最好是每次只添加几个相关的项（如只添加内存计数器），这样得到的图表比较有意义。

关于性能监视器的详细全面的介绍超出了本章的范围。建议去查阅它的其他功能，如数据收集器集和更改图表显示的特性。有很多详细描述这些功能的文档。

性能监视器的功能多种多样，应使其成为形成系统基准和随时间记录系统行为的最佳选择，可以把它当成实时诊断工具使用。



如果已经使用了可靠性或性能监视器，你可能注意到还有一个很少被提及的特性——资源概述。它是可靠性和性能监视器的默认视图，提供 CPU、磁盘、网络和内存的动态性能图，图下面的详细信息面板提供了以上四个方面的相关信息。该报告扩展自任务管理器的性能图，并为 Windows 操作系统的性能监控和诊断提供了另一个参考来源。

通过对微软 Windows 性能监视器的简单介绍，应该能够让你明白微软 Windows 平台很难进行监控，而且缺少监控工具的说法只是一个谎言。这些监控工具很全面（有些人甚至会说太多了），提供了系统数据的各种视图。

## 预防性维护监控

到目前为止，我们讨论的技术都是提供系统状态的快照。然而，大多数人认为监控通常是自动化的任务，记录统计数据以发现异常。当异常被发现时，向管理员（或管理员组）发出警报，让管理员知道系统可能出了问题。这将使被动监控系统状态变成一个主动性任务。

大量第三方工具将监控、报告和预警结合成易用的接口，甚至有针对于整个基础设施的监控和预警系统。比如，Nagios 能够监控整个 IT 基础设施并为异常设立预警。

当然还有一些监控和预警系统，用作操作系统或数据库系统的一部分或附属品。我们将在第 16 章介绍 MySQL 企业监控器。

## 小结

关于性能优化和安全监控有很多参考资料。本章介绍了系统监控的入门知识，虽然并不全面，但是简单介绍了监控操作系统和服务器性能的工具、技术和概念。下一章将开始介绍 MySQL 系统的监控任务，并讨论一些通用的实例以帮助 MySQL 系统保持在最佳性能状态运行。

“Joel!”

Joel 熟悉这个声音和语调。他的老板正朝他的办公室走来，准备分配另一个路过任务。他转过头看到老板正好走进他的办公室。“你看到 Sally 那封关于运行慢的邮件了没？”

Joel 想起来 Sally 是他的一个同事，曾经写了封邮件询问为什么她的应用程序运行得很慢。简单检查过后，他发现问题不在内存和磁盘上，因为有大量的内存和磁盘空间。

“是的，我正在调查这个问题。”

“把它当作你的首要任务。营销部门需要在截止日期之前产生季度销售报表。有发现了告诉我。”老板点了下头然后走开了。

Joel 叹了口气，继续检查 CPU 使用率报告，一边思考着怎样向非技术人员描述这个技术问题。



# 监控MySQL

Joel 觉得今天好多了，因为一切都进行得很顺利：服务器的性能测定结果看起来不错，用户投诉也在下降。他成功地对服务器进行了重新配置，使其性能大为提高。只有一个应用程序性能仍然不好，但他肯定这不是硬件或操作系统问题，倒更像是查询语句写得不好导致的。于是，他给老板写了封电子邮件，解释了他的发现，并且说明他正在处理遗留问题。

Joel 听到办公室外传来一阵急促的脚步声。他本能地朝门口看去，等待着老板那熟悉的身影出现。Summerson 先生只是路过，一句话都没说只是向他点了下头，把他吓了一跳。

他耸耸肩，继续看邮件。正在这时，一个主题写着“高优先级”的新消息出现了。是老板发过来的。Joel 觉得自己太过紧张了，松了一口气然后打开消息。看这封邮件的时候他仿佛能在脑海里听到老板的声音。

“Joel，这些报告写得很好。我特别喜欢你写的有关内存和磁盘性能的细节。我想要你写一个关于数据库服务器的类似报告。还有，了解一下有个开发人员在查询语句方面的问题。Susan 会把具体情况发给你。”

Joel 深深叹了口气，又一次翻开他最爱的 MySQL 书，学习更多有关监控数据库系统的知识。“希望这里有每个部件的详细介绍。”他咕哝着，知道自己需要恶补 MySQL 的高级功能了。

现在，我们已经知道监控的工作原理，以及如何让操作系统达到最高效率，那么如何知道 MySQL 服务器是否最高效地运行呢？或者，怎么知道它们不是高效运行的？

这一章我们首先介绍 MySQL 监控，包括 MySQL 的监控方法和分类。然后介绍如何监控数据库和提高数据库性能。最后给出提高数据库系统性能的最佳实践。

# 什么是性能

在开始讨论数据库性能及监控和优化 MySQL 服务器的常用最佳实践之前，最好先定义一下性能的含义。就本章而言，良好的性能被定义为能够满足用户的需求，也就是指系统按照用户期望的那样得当地运行，而低性能被定义成系统运行不佳。通常情况下，良好的性能意味着响应时间和吞吐量能够满足用户的期望。虽然可能看上去不太科学，但是专业的管理员都知道，衡量系统好坏的最佳指标就是用户满意度。

但这并不意味着不测量性能。相反，我们要而且必须要测量性能，才能知道有什么问题需要修正，什么时候进行修正，以及如何修正。而且，如果定期测量性能，甚至可以预测用户什么时候开始不高兴。用户并不在乎你是否将缓存命中率增加了三个百分点，打破了最高纪录。你可能会为这样的事情感到自豪，但是，与用户体验相比，指标和数字是没有意义的。

处理性能问题应该遵循一个非常重要的基本原理：除非有深思熟虑的计划，并且相当了解变更后的期望和后果，否则永远不要更改服务器、数据库或存储引擎的参数。更重要的是，如果没有评估变更的影响，永远不要做出改动。完全有可能发生这样的情况：你可能在短期内提高了服务器性能，但是长远来看却对性能产生负面影响。最后，总是应该查阅各方面的参考信息，包括参考手册等。

既然我们已经给出了严厉的警告，接下来讲述 MySQL 服务器和数据库的监控和提高性能这两方面的问题。



管理员监控 MySQL 几乎总是关注性能提升。性能固然重要，关系到用户需要等待多久才能执行查询。但是监控也会检查资源枯竭或资源的高需求量带来的超时及访问服务器的其他故障问题。

## 381 MySQL 服务器监控

管理 MySQL 服务器属于应用程序监控的范畴。因为大部分性能参数是由 MySQL 软件产生的，而不是主机操作系统的一部分。前面说过，应该总是同步监控基础操作系统和 MySQL，因为 MySQL 对主机操作系统的性能很敏感。

在线 MySQL 参考手册中有一整章介绍监控和性能提升的各方面问题，名为“优化”(<http://bit.ly/mysql-opt>)。我们不再重复介绍参考文档上的知识，而是讨论监控 MySQL 服务器的一般方法，并介绍几种不同的监控工具。

本节详细介绍监控 MySQL 服务器。首先简单介绍如何更改和监控系统的行为，然后讨论如何为了诊断性能问题和形成性能基准而监控。此外，还将介绍诊断性能问题的最佳实践，以及监控 MySQL 存储引擎方面的知识，这方面的知识在其他参考资料上很少被提及。

## 如何显示 MySQL 性能

有两种机制来管理和监控 MySQL 服务器的行为。使用服务器变量来控制其运行情况，使用服务器状态变量读取其行为配置和关于功能和性能的统计信息。

有很多变量用于配置服务器。有些变量只能在启动时设置（被称为启动选项，也可以在选项文件中设置）。其他变量可以被设置成全局级（对于所有连接有效）、会话级（仅对单个连接有效），或同时适用于全局和会话两个级别。

使用以下命令可读取服务器变量：

```
SHOW [GLOBAL | SESSION] VARIABLES;
```

使用以下命令可更改非静态（只读）变量（可以用逗号分隔符在一个命令行同时设置多个变量）：

```
SET [GLOBAL | SESSION] variable_name = value;  
SET @@global. | @@session. | @@]variable_name = value;
```



会话变量只在当前连接中有效，在连接关闭时被重置。

使用以下命令读取状态变量。前两条命令显示所有本地或会话范围的变量值（默认是会话变量），第三条命令显示全局范围的变量：

382

```
SHOW STATUS;  
SHOW SESSION STATUS;  
SHOW GLOBAL STATUS;
```

下一节将讨论如何使用及何时使用这些命令。

查看服务器的信息及其运行状态有两个重要的命令，即 `SHOW VARIABLES` 和 `SHOW STATUS`。涉及很多变量（仅状态变量就超过 290 个），这些变量通常按照字母顺序列出，并根据功能分组。不过，有时候变量并不完全整洁有序。使用 `LIKE` 语句过滤关键字能够

产生系统特定方面的相关信息。例如，`SHOW STATUS LIKE '%thread%'` 显示与线程运行相关的所有状态变量。

## 性能监控

MySQL 的性能监控是前面命令的应用，即设置和读取系统变量及读取状态变量。`SHOW` 和 `SET` 命令是仅有的两个可以用于监控 MySQL 服务器的工具。

事实上，有几个工具可以用于监控 MySQL 服务器。而标准发行版中的监控工具却很有限，因为它们都是控制台工具，包括 MySQL 客户端的特殊命令（如 `SHOW STATUS`）和命令行实用工具（如 `mysqladmin`）。



MySQL 客户端工具有时被称为 MySQL 监控器，但是不要将它与监控工具混为一谈。

还有一些 GUI 工具使监控变得更容易，如果你有这方面的需求，可以考虑选择这种工具。还可以下载 MySQL GUI 工具，其中包括高级工具，这些高级工具可以用于监控系统、管理查询和从其他数据库系统迁移数据。

下面首先介绍如何使用这些 SQL 命令，然后讨论 MySQL 工作台工具。还将简单介绍最容易被忽视的管理工具之一——服务器日志。

**383** 有些专业的管理员在管理服务器时，第一时间考虑的主要工具也许就是服务器日志。虽然服务器日志不如性能监控工具那么重要，但是它们在诊断性能问题时也是非常重要的。

## SQL 命令

所有的 SQL 监控命令都可以认为是 `SHOW` 命令的变体，显示系统及其子系统的内部信息。例如，监控复制时很有用的一对命令是 `SHOW MASTER STATUS` 和 `SHOW SLAVE STATUS`。这一章我们将深入讨论这些命令。



很多命令可以直接通过查询 `INFORMATION_SCHEMA` 表来实现。参考在线 MySQL 参考手册，了解更多有关 `INFORMATION_SCHEMA` 数据库的详细信息和功能。

`SHOW` 命令的形式很多，下面列出了监控 MySQL 服务器时最常用的 SQL 命令：



## SHOW INDEX FROM *table*

描述表中的索引，确定是否使用了正确的索引。

## SHOW PLUGINS

列出所有已知插件，显示插件的名称和当前状态。MySQL 最新发行版中的存储引擎是以插件形式实现的。使用这个命令获取当前可用插件及其状态的快照。虽然与性能监控并无直接联系，但有些插件提供了系统变量。了解安装了哪些插件能够确定哪些插件特定的变量是可访问的。

## SHOW [FULL] PROCESSLIST

显示系统上运行的所有线程（包括处理客户端连接的线程）数据。这个命令与主机操作系统的进程命令类似。显示的信息包括命令执行时的连接数据、命令运行了多长时间和当前状态。像操作系统命令一样，它可以诊断响应差的（太多线程）、僵尸进程（无响应或长时间运行），或者甚至诊断连接问题。在处理低性能或无响应的线程时，使用 KILL 命令终止这些线程。默认显示当前用户的进程。使用 FULL 关键字显示所有进程。



必须有全局 SUPER 权限才能查看运行在系统上的所有进程。

384

## SHOW [GLOBAL | SESSION] STATUS

显示所有系统变量的值。相对其他命令而言，这个命令可能用得更多。使用这个命令读取服务器的所有统计信息。与 GLOBAL 或 SESSION 关键字结合使用，可以选择性地只查看全局变量的统计信息或会话变量的统计信息。

## SHOW TABLE [FROM *db*] STATUS

显示给定数据库的表的详情，包括存储引擎、排序规则 (collation)、创建数据、索引数据和行统计信息。与 SHOW INDEX 命令结合使用，能够在诊断低性能查询时检查表信息。

## SHOW [ GLOBAL | SESSION] VARIABLES

显示系统变量。通常是服务器的配置选项，虽然它不显示统计信息，但是在确定当前配置是否已被更改或某些选项是否被设置时，查看变量是非常重要的。有些变量是只读的，只能通过配置文件或命令行在启动的时候更改，而有些变量可以在全局或本地范围内设置。与 GLOBAL 或 SESSION 关键字结合使用，能够选择性地查看全局变量或会话变量。

## 限定 SHOW 命令的输出结果

MySQL 中的 SHOW 命令功能很强大。但是显示的信息常常太多了，尤其是 SHOW STATUS 和 SHOW VARIABLES 命令。

使用 LIKE *pattern* 从句可查看较少的信息，只显示那些匹配模式的行。最常用的例子是使用 LIKE 语句查看变量的某个子集，如复制或日志。与 SELECT 查询一样，在 LIKE 语句中使用标准 MySQL 正则表达式符号和控制符。

例如，显示名称包含“log”的状态变量：

```
mysql> SHOW SESSION STATUS LIKE '%log%';
```

Variable_name	Value
Binlog_cache_disk_use	0
Binlog_cache_use	0
Com_binlog	0
Com_purge_bup_log	0
Com_show_binlog_events	0
Com_show_binlogs	0
Com_show_engine_logs	0
Com_show_relaylog_events	0
Tc_log_max_pages_used	0
Tc_log_page_size	0
Tc_log_page_waits	0

11 rows in set (0.11 sec)

与存储引擎相关的命令如下：

**SHOW ENGINE *engine\_name* LOGS**

显示指定存储引擎的日志信息，该信息显示依赖于存储引擎，而且在优化存储引擎时非常有用。有些存储引擎不提供这些信息。

**SHOW ENGINE *engine\_name* STATUS**

显示指定存储引擎的状态信息，该信息依赖于存储引擎。有些存储引擎显示的信息比其他存储引擎多。例如，InnoDB 存储引擎显示几十个状态变量，NDB 存储引擎只显示几个状态变量，而 MyISAM 存储引擎则不显示任何信息。这个命令提供了查看特定存储引擎的统计信息的基本方法，在优化某些存储引擎时非常重要（如 InnoDB）。



SHOW ENGINE 命令的旧同义形式 (SHOW engine LOGS 和 SHOW engine STATUS) 已不再使用。而且, 这些命令只在某些引擎中显示信息, 比如 InnoDB 和 Performance\_Schema。

## SHOW ENGINES

显示 MySQL 发行版所有已知的存储引擎列表及其状态 (如存储引擎是否启用)。这个命令有助于决定在数据库上使用何种存储引擎, 以及复制时是否在 master 和 slave 上存在相同的存储引擎。

与 MySQL 复制相关的命令如下:

SHOW BINLOG EVENTS [IN log\_file] [FROM pos] [LIMIT offset row\_count]

显示被记录到二进制日志中的事件。指定要审查的日志文件 (如果没有 IN 从句则默认使用当前日志文件), 将输出限定为某个特定位置之后的所有事件, 或者某个偏移位置之后的行。这是用于诊断复制问题的主要命令, 如果某个事件导致复制中断或错误, 这个命令非常有用。

◀ 386



如果不使用 LIMIT 语句, 而且服务器已经运行了一段时间并记录了日志, 那么将会得到一个很长的输出结果。如果需要查看大量事件, 应该考虑使用 mysqlbinlog 实用工具代替这个命令。

## SHOW BINARY LOGS

显示服务器上的二进制日志列表, 获取过去和当前 binlog 文件名的相关信息。每个文件的大小也被显示出来了。它是诊断复制问题的另一个有用的命令, 通过这个命令为 SHOW BINLOG EVENTS 指定日志文件, 从而减少在诊断问题时必须查看的数据量。SHOW MASTER LOGS 是它的同义词。

SHOW RELAYLOG EVENTS [IN log\_file] [FROM pos] [LIMIT offset row\_count]

在 MySQL 5.5.0 中可用, 这个命令和 SHOW BINLOG EVENTS 命令的功能一样, 只不过是对 slave 上的中继日志而言的。如果不提供日志文件名, 该命令将显示第一个中继日志中的事件。这个命令在 master 上运行无效。

## SHOW MASTER STATUS

显示 master 的当前配置。显示当前二进制日志文件、文件的当前位置和所有排他性或包容性的复制设置。当连接或重新连接 slave 时使用这个命令。

## SHOW SLAVE HOSTS

显示那些通过 --report-host 选项连接到 master 的 slave 列表。根据这个列表可以

确定哪些 slave 连接到 master 上。

## SHOW SLAVE STATUS

显示复制中 slave 的系统状态信息。这是追踪 slave 性能和状态的主要命令。这个命令显示了大量有关 slave 健康状态维护的重要信息。可参见第 3 章关于这个命令的信息。

示例 11-1 给出了 SHOW VARIABLES 命令及其输出的例子。

### 387 示例 11-1: 显示线程状态变量

```
mysql> SHOW VARIABLES LIKE '%thread%';
```

Variable_name	Value
innodb_file_io_threads	4
innodb_read_io_threads	4
innodb_thread_concurrency	0
innodb_thread_sleep_delay	10000
innodb_write_io_threads	4
max_delayed_threads	20
max_insert_delayed_threads	20
myisam_repair_threads	1
pseudo_thread_id	1
thread_cache_size	0
thread_handling	one-thread-per-connection
thread_stack	262144

12 rows in set (0.00 sec)

这个例子不仅显示了线程管理的状态和变量，还显示了 InnoDB 存储引擎的线程管理。虽然有时候得到的结果比你想象的多，但是可以使用 LIKE 语句查找需要的变量。

监控 MySQL 服务器最具挑战性的问题之一是确定哪些变量需要更改以及哪些变量需要监控。在线 MySQL 参考手册上有大量关于这方面的有价值的信息。

为了说明 MySQL 服务器的监控功能，我们来讨论控制查询缓存（query cache）的变量。如果应用数据使用的是 MyISAM 存储引擎，查询缓存是 MySQL 最重要的性能特征之一。它允许服务器在内存中缓存频繁使用的查询语句和查询结果。因此，查询运行得越频繁，其查询结果就越有可能从缓存中读取，而不是重新检查索引和表来获取数据。显然，从内存中读取数据比每次都从硬盘上读取数据要快很多。如果读数据频率比写（更新）频率高得多，这就可以提高性能。



每次运行查询，这个查询将被放入缓存并且有生命周期，该生命周期由它最近被使用的情况（旧查询首先被回收）和查询缓存可用的内存量来决定，另外，还有很多事件可以使查询在缓存中失效（删除）。

这里我们列举了部分事件：

- 数据或索引的变更。
- 同一个查询的微小区别会产生不同的结果集，从而导致缓存未命中。因此，使用标准的查询访问公共数据是非常重要的。后面你会看到视图在这个问题上起的作用。
- 当查询从临时表（不被缓存）中获取数据时。
- 能够使内存中的查询无效的事务事件（如 COMMIT）。

检查 `have_query_cache` 变量，确定安装在 MySQL 中的查询缓存是否被配置和是否可用。这是一个全局的系统变量，但是它是只读的。使用其中一个变量来控制查询缓存。示例 11-2 显示了查询缓存的服务器变量。

示例 11-2: 查询缓存的服务器变量

```
mysql> SHOW VARIABLES LIKE '%query_cache%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| have_query_cache | YES |
| query_cache_limit | 1048576 |
| query_cache_min_res_unit | 4096 |
| query_cache_size | 33554432 |
| query_cache_type | ON |
| query_cache_wlock_invalidate | OFF |
+-----+-----+
6 rows in set (0.00 sec)
```

如你所见，影响查询缓存的做法有若干。最值得注意的是设置 `query_cache_size` 变量临时关闭查询缓存。这个变量能够设置查询缓存的可用内存大小。如果设为 0，则立即关闭查询缓存，并将缓存中的所有查询删除。这与 `have_query_cache` 变量无关，`have_query_cache` 变量仅仅表明查询缓存是否可用。而且，设置 `query_cache_type = OFF` 还不够，因为这并不会释放查询缓存的缓冲区。必须同时将 `query_cache_size` 设为 0，才能完全关闭查询缓存。更多关于配置查询缓存的信息，请参考在线 MySQL 参考手册中的“查询缓存配置”一节。

检查多个状态变量，观察查询缓存的性能，如示例 11-3 所示。

示例11-3: 查询缓存的状态变量

```
mysql> SHOW STATUS LIKE '%Qcache%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Qcache_free_blocks | 0 |
| Qcache_free_memory | 0 |
| Qcache_hits | 0 |
| Qcache_inserts | 0 |
| Qcache_lowmem_prunes | 0 |
| Qcache_not_cached | 0 |
| Qcache_queries_in_cache | 0 |
| Qcache_total_blocks | 0 |
+-----+-----+
8 rows in set (0.00 sec)
```

这里我们看到了 MySQL 服务器微妙的不一致性。通过以 `query_cache` 开头的变量控制查询缓存, 而状态变量是以 `Qcache` 开头的。虽然这个不一致性是故意设计的 (为了区分服务器变量和状态变量), 但像这样奇怪的设计却使搜索变得复杂。

查询缓存在管理、配置和性能监控上还有很多玄妙之处, 所以查询缓存是解释 MySQL 服务器监控的绝佳例子。

例如, 应该使用 `FLUSH QUERY CACHE` 命令定期重整查询缓存。这样不会删除内存中的结果, 而是重新组织内存以更好地使用内存。

关于这个话题以及查询缓存的玄妙之处不做单独章节进行介绍, 本章描述的实践是结合 MySQL 的功能通用的。当然, 如果想了解更多详情, 请自行阅读在线 MySQL 参考手册。

## mysqladmin 实用工具

`mysqladmin` 命令行工具是命令行工具套件中的重量级工具。这个工具有很多选项和工具 (称为“命令”)。在线 MySQL 参考手册简单讨论了 `mysqladmin` 工具。本节将讨论监控 MySQL 服务器的选项和工具。

这个实用工具是从命令行启动运行的, 所以管理员可以编写一些操作脚本, 这比直接运行 SQL 命令要容易很多。事实上, 有些第三方监控工具将 `mysqladmin` 和 SQL 命令结合起来以收集信息, 并以其他形式显示。

必须指定连接到某个运行服务器的连接信息 (用户、密码、主机等)。下面列出了常用的命令 (你会发现, 大部分命令与相应的 SQL 命令等效):

## status

简要显示服务器状态信息，包括正常运行时间、线程（连接）数、查询数和常规统计数据。这个命令提供服务器健康状况的一个快照。

## extended-status

显示系统统计信息的完整列表，与 SQL SHOW STATUS 命令类似。

## processlist

显示当前进程的列表，并与 SQL SHOW PROCESSLIST 命令类似。

## kill thread id

杀死某个指定的线程。它与 processlist 结合使用，可以帮助管理失控或挂起的进程。

## variables

显示系统服务器变量和值。与 SQL SHOW VARIABLES 命令类似。

还有很多其他的选项和命令没有在这里罗列出来，包括在复制过程中启动和停止 slave 的命令，以及管理各种系统日志的命令。

*mysqladmin* 的最佳功能之一是能够对比不同时间的信息。--sleep *n* 选项告诉实用程序每 *n* 秒执行一次指定的命令。例如，使用如下命令在本地主机上每 3 秒刷新一次进程列表：

```
mysqladmin -uroot --password processlist --sleep 3
```

这个命令将一直执行，直到使用 Ctrl+C 组合键关闭工具为止。

也许最强大的功能是能够对比扩展状态命令的结果。使用 --relative 选项将先前的执行值与当前值相比较。例如，使用以下命令查看系统状态变量先前的值和当前值：

```
mysqladmin -uroot --password extended-status --relative --sleep 3
```

还可以合并命令从而同时获得多份报告。例如，使用以下命令同时查看进程列表和状态信息：

```
mysqladmin --root ... processlist status
```

*mysqladmin* 工具还有很多其他用处。可以用来关闭服务器、刷新日志、ping 服务器、在复制中启动和关闭 slave，以及刷新权限表。关于 *mysqladmin* 工具的更多信息请查看在线 MySQL 参考手册中的“mysqladmin——管理 MySQL 服务器的客户端”一节。图 11-1 显示了一个没有负载的系统的输出样本。

Id	User	Host	db	Command	Time	State	Info
51	root	localhost:52264		Sleep	423		
52	root	localhost:52265		Sleep	426		
204	root	localhost		Query	0		show processlist

Uptime: 533 Threads: 3 Questions: 50 Slow queries: 0 Opens: 17 Flush tables: 1  
Open tables: 10 Queries per second avg: 0.93

图11-1: mysqladmin进程和状态报告样例

## 391 MySQL 工作台

MySQL 工作台应用是基于工作站的 GUI 管理工具。MySQL 工作台，这里我们简称为工作台，在 MySQL 网站上可供下载，由 MySQL 社区版（GPL）或标准版提供。标准版绑定了 MySQL 企业版的一些功能。

MySQL 工作台的主要功能有：

- 服务器管理器
- SQL 开发
- 数据建模
- 数据库迁移向导

接下来的几节我们将详细讨论服务器管理，并简单介绍一下 SQL 开发。数据建模超出了本书的范畴，但如果想要实现数据库模式的配置管理，建议研究一下工作台文档中提到的功能。数据库迁移向导用来自动化从其他数据库系统迁移数据库模式和数据，包括 Microsoft SQL Server 2000/2005/2008/2012，PostgreSQL 8.0 及其以后版本，Sybase Adaptive Server Enterprise 15.x 及其以后版本等。这个工具非常方便，能够快速简单地导入 MySQL。



MySQL 工作台代替了旧的 MySQL GUI 工具，包括 MySQL 管理器、MySQL 查询浏览器和 MySQL 迁移工具箱。

392 启动工作台时，主界面显示 3 个部分，分别是 SQL 开发、数据建模和服务器管理器（参见图 11-2）。每个部分下方有一些链接能够开始不同的工作。数据库迁移功能通过“数据库迁移...”菜单访问。





图11-2: MySQL工作台主界面

可以在任何平台上使用工作台，客户端可以访问一个或多个服务器。当需要监控网络上的多个服务器时，有这个工具就方便多了。

关于安装和配置的更多信息可参考在线 MySQL 工作台文档。

## MySQL 服务器管理器

服务器管理器能够查看和更改系统变量，管理配置文件，检查服务器日志，监控状态变量，甚至以图形方式查看某些重要的性能数据。此外，它还有一组完整的管理选项，能够管理用户和查看数据库配置。最初这个工具是要替代 *mysqladmin* 的，不过今后我们可能同时需要这两个工具。

要使用服务器管理器，首先必须定义一个 MySQL 服务器实例。单击“新建服务器实例”链接，然后按照步骤建立一个连接服务器的新实例（连接）。这个过程会尝试连接服务器，验证提供的参数，确保这是一个有效的实例。一旦实例建立以后，就会以方框的形式显示在主窗口的“服务器管理器”的下方。

要管理服务器，从列表中选择一个服务器实例，然后单击“服务器管理器”，就会看到图 11-3 所示的新窗口。

393

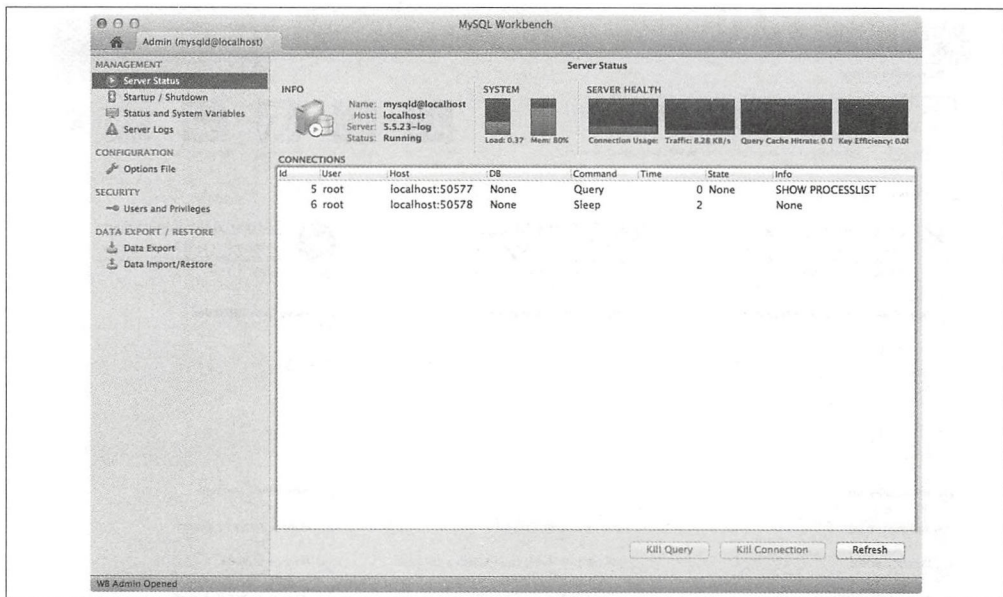


图11-3: 服务器管理器窗口

注意,左边有4个部分:管理、配置、安全和数据导入/导出。现在我们简单解释一下每一项。

**管理。**管理组中的工具能够提供服务器状态的概述,启动和停止服务器,查看系统和状态变量,以及查看服务器日志。

394



本书第一版中,我们介绍了 MySQL 管理器应用,它能够以图形显示内存使用、连接及更多信息。MySQL 工作台没有这个功能,而 MySQL 企业监视器应用包含这个功能。相比弃用的 MySQL 管理器工具中的功能,图形显示功能高级很多。

我们来看图 11-3 中所示的服务器状态。注意,有一个服务器负载及其内存使用率的小图。右边分别是连接使用率、网络流量、查询缓存命中率和关键效率的图形。通过这些图,可以迅速了解服务器的状态。如果任何一个图中有异常高的值(或者异常低的值,这种情况不常见),则在问题尚未严重之前,以此为线索检查性能问题。

如果想要工具提供更细粒度的系统状态和健康状况等,可能需要 MySQL 企业监视器应用,这将在第 16 章讨论。

启动和关闭工具就能启动或关闭服务器实例。该工具还能展示来自服务器的最近的消息,应该随工具一起启动或关闭服务器。

状态和系统变量工具是管理组中最常用的。图 11-4 给出了这个工具的截图。通过类别选

择或者检索某个短语（类似 LIKE '%test%'），就能查看状态变量了。

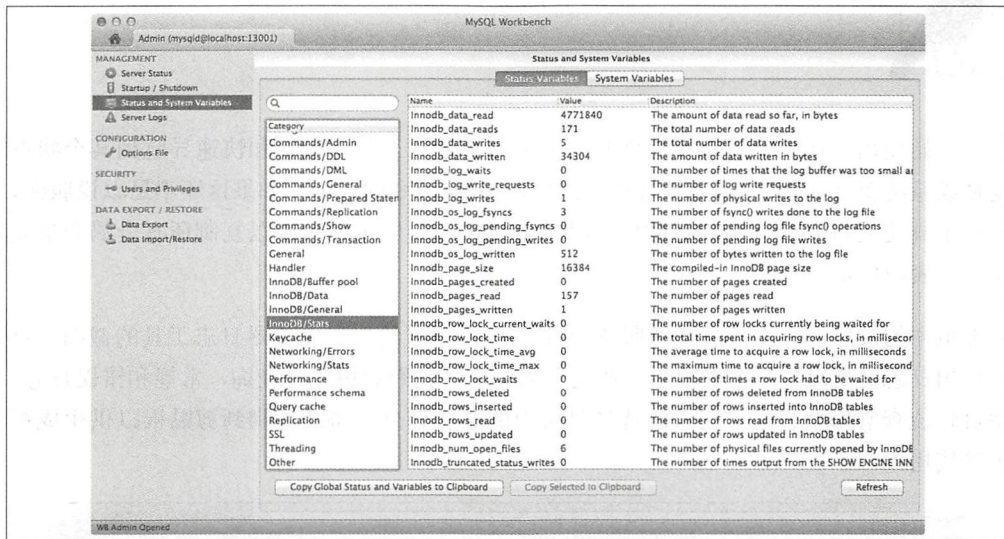


图11-4：状态变量

“系统变量”选项卡也有同样的检索功能。图 11-5 给出了系统变量工具的截图。你会看到，定义了大量类别，通过这些类别能够迅速找到需要查看的内容。任何前缀为 [rw] 的变量都是可读写的，所以在运行时可能会被管理员更改。

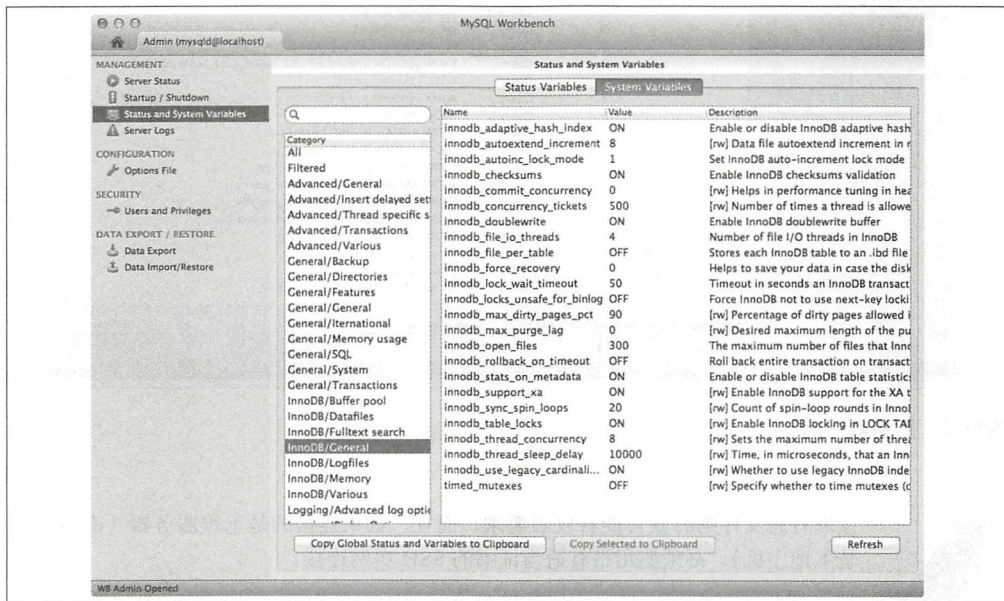


图11-5：系统变量





管理员账户必须具备 SUPER 权限。

396 一旦开始使用工作台，就会发现你常常需要用到这些工具。检索和快速导航到某个状态变量或系统变量，能够节省大量敲单词或执行 SHOW 命令的时间。如果这还不足以说服你，这些工具还能将变量复制到剪贴板，以供生成报告等使用。你可以复制所有全局变量或仅在列表中显示的变量。

管理组中的最后一个工具是查看服务器日志。图 11-6 给出了服务器日志工具的截图。每种启用日志类型对应一个选项卡。在这个例子中，我们启用了慢查询、常规和错误日志。按页轮流查看每个日志。还可以选择日志中的某些部分，将其复制到剪贴板以供生成报告等使用。

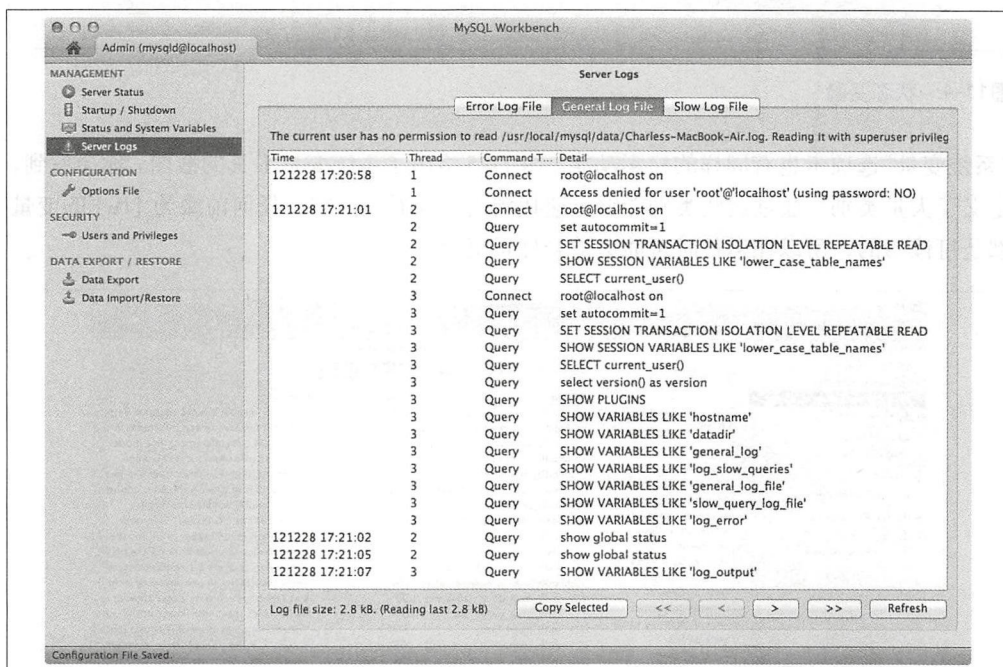


图11-6：服务器日志



读取日志文件的时候可能有限要求。而且，如果连接的是远程服务器（而不是本地主机），必须使用带有适当证书的 SSH 实例连接。



你会发现，管理 MySQL 服务器的图形工具能够简化重复性的基本任务。

配置。第二组是管理配置文件的工具。图 11-7 给出了选项文件工具的截图。不仅可以看到配置文件中设置了哪些选项，还能够更改它们的值并保存新值。马上细说。

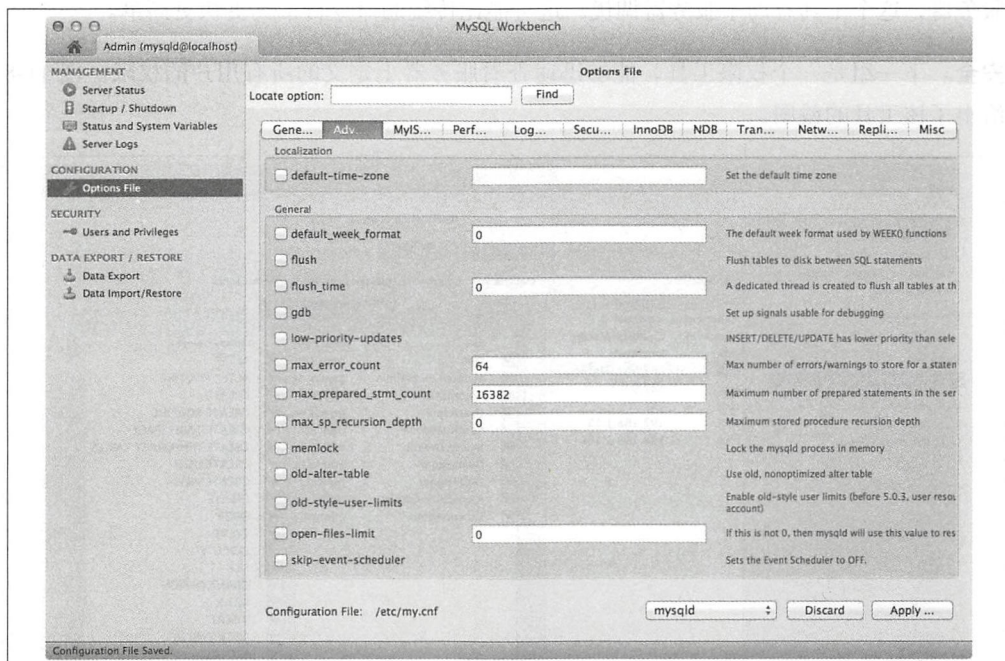


图11-7：选项文件



如果修改文件，用户账号必须具备操作系统级的写权限。

上面一组选项卡给出了若干类别，包括常规、高级、MyISAM、性能、日志文件、安全、InnoDB、NDB、事务、网络、复制和其他。该工具囊括了该版本服务器已知的所有服务器选项，使用分类使得寻找和设置配置文件条目更加容易。每个选项右侧还提供了一个短文本帮助提示。

设置选项时首先需要勾选复选框，表明选项应该出现在文件中。而且，如果选项需要设定值，则在文本框中输入值或更改已有的值。一旦设置完全部选项，单击“Apply”按钮使其生效。单击“Apply”按钮时，会打开一个对话框显示文件变动概览。选择取消或应用这些变更，也可以从对话框中看到文件的全部内容。单击“Apply”按钮就会保

存这些变更，在下一一次启动服务器的时候生效。

这个工具还有一个强大的功能。注意，底部有一个名为“mysqld”的下拉框，用来设置正在编辑的配置文件的某一段，因此用这个工具可以为某些应用修改选项。再加上重启服务器，这个工具有助于服务器调优。这可能比传统的命令行工具更快更简单。

安全。下一组是一个权限工具，能够迅速查看服务器上定义的所有用户的权限。图 11-8 给出了该工具的截图。

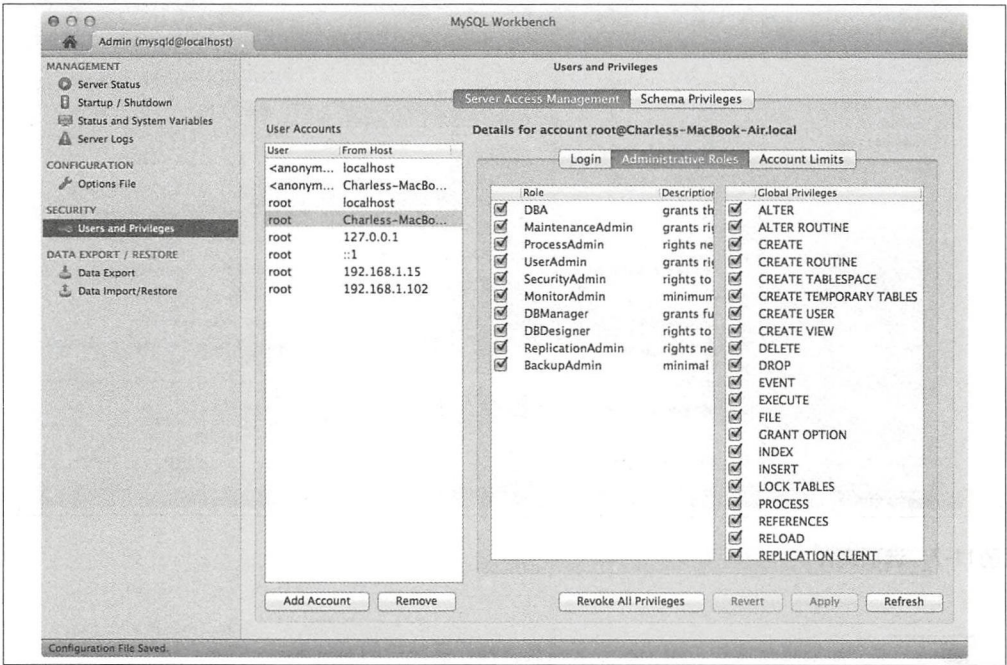


图11-8：权限

使用这个工具有助于诊断访问问题，优化权限设置使得用户访问最小化。而且，这个工具还可以更改指定用户的权限，单击复选框切换访问即可（没有勾选说明用户没有权限）。做完更改以后，单击“Apply”按钮实现相应的变更。

数据导出 / 恢复。最后一组工具封装了 *mysqldump* 的基本的数据导出和导入功能。严格来说，这个工具不属于监控，但工具集中还是要要有这个功能。例如，将数据从一个服务器复制或导出到另一个服务器，进而分析性能相关的查询问题。

可以选择导出整个数据库或任意组合的对象。图 11-9 给出了导出功能的截图示例。

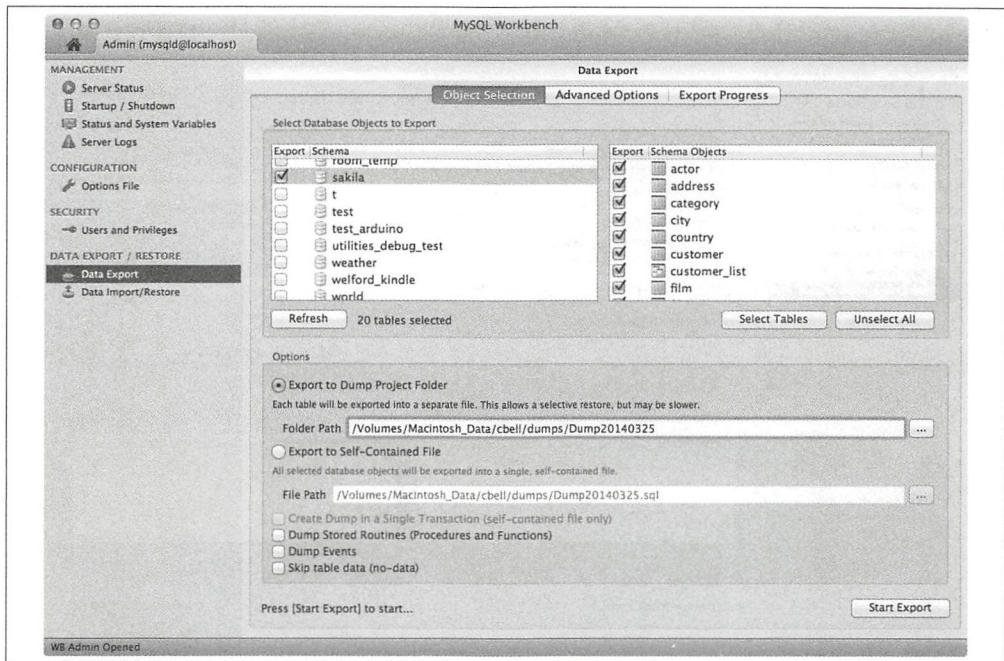


图11-9: 数据导出

将所有对象和数据转储到单个文件中，或者指定一个项目文件夹，每张表保存为单独的 *.sql* 文件，其内容是表创建和插入数据的 SQL 语句。选择一个选项，以及想要导出的数据库和表，然后单击“Start Export”按钮，就会运行相应的 *mysqldump* 命令。打开一个概览对话框显示操作进度，以及执行导出所使用的命令。可以保存这些命令方便在脚本中使用。

还可以选择导出过程和函数、事件，或者部分导出数据（只导出表结构）。如果使用的是 InnoDB 数据库，还可以告诉工具使用单个事务以避免长时间锁定表。这时，工具告诉 *mysqldump* 使用 InnoDB 的一致性快照功能来锁定表。

400

通过数据导入 / 恢复工具（参见图 11-10）可实现数据导入。选择想要导入的导出文件夹或文件，以及目标数据库（模式）。

如果想要导出到项目文件夹，还要选择想要导入哪些文件（表），以便执行选择性恢复。同导出工具一样，执行导入也会打开一个对话框，显示导入进度和 *mysqldump* 命令。



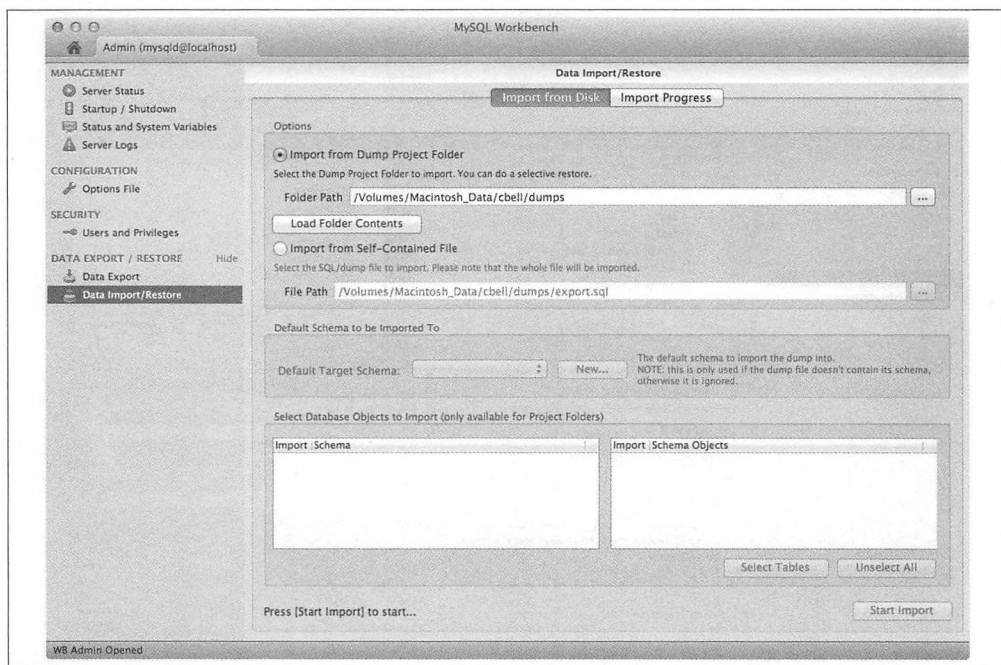


图11-10: 数据导入

## SQL 开发

工作台中另一个 GUI 工具是 SQL 编辑器。同样，这个工具本身也不是一个监控工具，但是你会发现，它为构建复杂 SQL 语句提供了健壮的环境。

401 从主窗口访问这个工具。选择一个实例，然后单击“打开连接开始查询”。图 11-11 给出了一个示例截图。

使用 SQL 编辑器可以构建查询，并以图形方式执行它们。返回的结果集以电子表格的形式显示。SQL 编辑器支持在整个结果集上垂直滚动和水平滚动，还可以更改列的大小以更好地查看数据。很多用户觉得这个工具比传统的 *mysql* 命令行客户端简单方便多了。

对任意查询执行 EXPLAIN 命令，将以图形显示性能相关的功能和管理员添加的值。图 11-12 所示的示例显示了 *world* (InnoDB) 数据库某个查询上的解释。稍后我们再详细讨论这个问题。

从这个 SQL 编辑器例子应该能看出来 GUI 的实用价值。输入任何查询执行这个查询，然后选择“查询”菜单中的“解释查询”项，查看这个查询执行的解释。



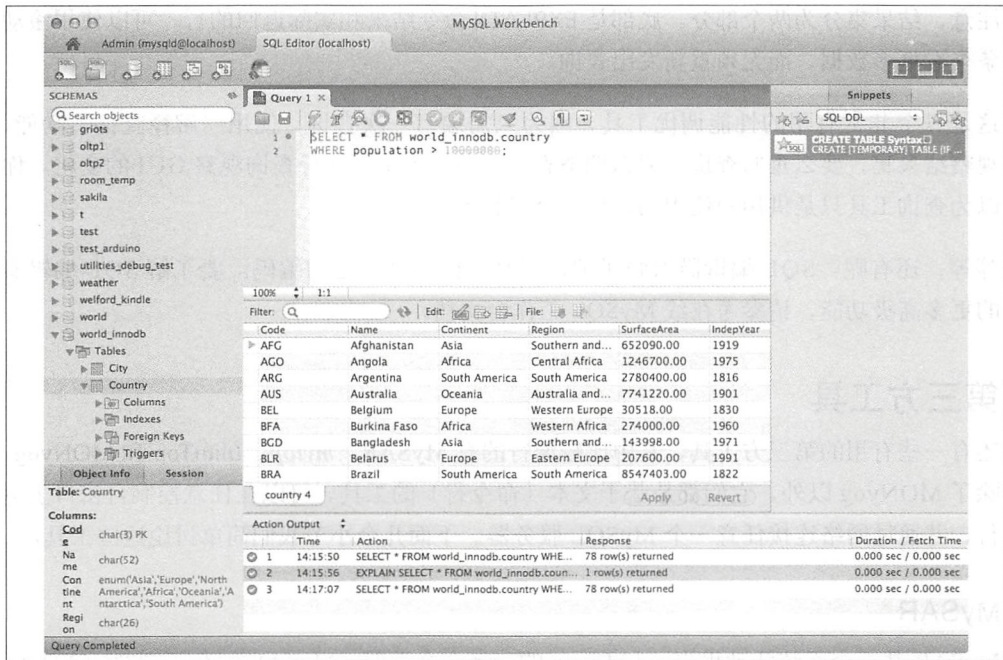


图11-11: SQL编辑器

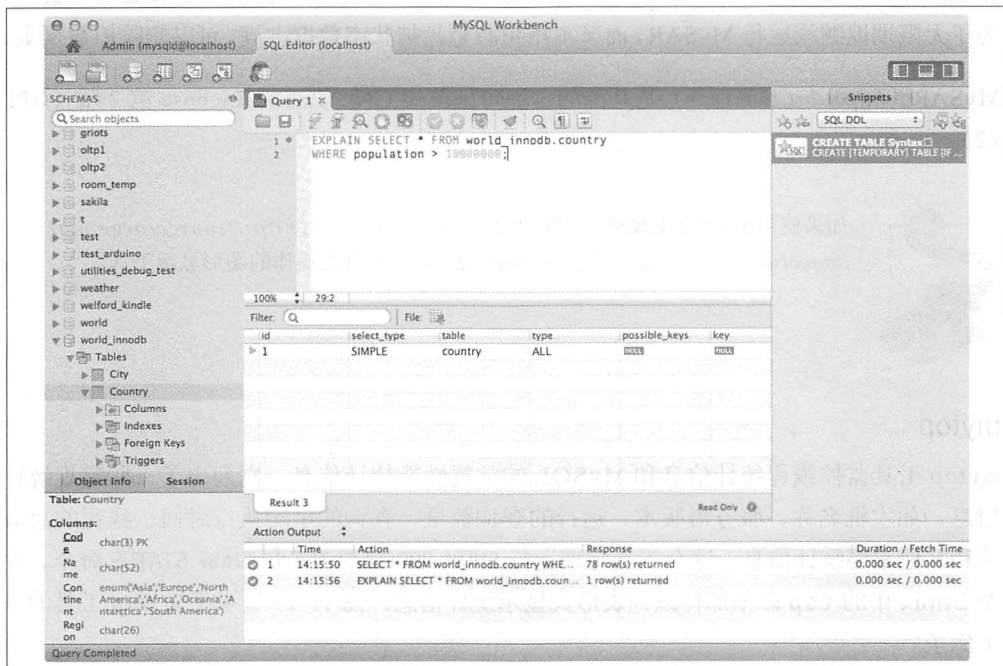


图11-12: SQL编辑器: 结果集视图

402 注意，结果集分为两个部分。底部是 EXPLAIN 命令结果和实际返回的行，可以使用滚动条查看更多数据，而无需重新发出查询。

这是一个非常有用的性能调优工具，因为只需编写一次查询，使用“解释查询”功能，观察结果集，要么重写查询，要么调整索引，然后再重新执行查询观察 GUI 的变化。你以为查询工具只是供用户使用的，但这个工具不是。

等等，还有呢。SQL 编辑器增强了编辑工具，比如带颜色的编码。要了解 SQL 编辑器的更多高级功能，请参考在线 MySQL 工作台文档。

## 第三方工具

还有一些有用的第三方工具，其中比较流行的有 MySAR、*mytop*、InnoTop 和 MONyog。除了 MONyog 以外，它们都是基于文本（命令行）的工具，可以在任意控制台窗口上运行，并通过网络连接任意一个 MySQL 服务器。下面几个小节我们简单讨论这些工具。

### 403 MySAR

MySAR 是一个系统活动报告，与 Linux 的 `sar` 命令类似。MySAR 综合了 `SHOW STATUS`、`SHOW VARIABLES` 和 `SHOW FULL PROCESSLIST` 命令的输出结果，并将这些结果存储在服务器上的 *mysar* 数据库中。可以使用各种方法配置这个数据集，包括限定收集哪些数据。为了无限期地继续运行 MySAR，而又不用担心磁盘被状态数据占满，可以删除旧数据集。

MySAR (<http://bit.ly/mysar>) 是开源的，许可证使用 GNU Public License 第 2 版 (GPL v2)。



如果使用 `sar` 收集数据，可能还要看看 `ksar` 工具 (<http://sourceforge.net/projects/ksar/>)。ksar 工具是在 `sar` 的结果集上进行操作的图形显示工具。

### mytop

*mytop* 工具监控线程统计信息和 MySQL 的常规性能统计信息。它列出了一些常规统计信息，如主机名称、服务器版本、运行的查询数量、查询的平均执行时间、线程的总数量和其他重要统计信息。这个工具定期运行 `SHOW PROCESSLIST` 和 `SHOW STATUS` 命令，并像 Linux 中的 `top` 命令那样以列表形式显示统计信息。图 11-13 显示了 *mytop* 工具的一个例子。

mytop						
MySQL on localhost (5.6.13-log) up 0+01:59:56 [16:45:00]						
Queries: 129.0 qps: 0 Slow: 0.0 Se/In/Up/De(%): 00/00/00/00						
qps now: 0 Slow qps: 0.0 Threads: 4 ( 1/ 1) 00/00/00/00						
Key Efficiency: 84.0% Bps in/out: 0.4/ 88.3 Now in/out: 8.4/ 1.9k						
Id	User	Host/IP	DB	Time	Cmd	Query or State
6850	root	localhost		0	Query	show full processlist
6917	root	localhost:38019		251	Sleep	
6918	root	localhost:38020		251	Sleep	
5	ltd_user	localhost:57628	mysql	7177	Sleep	
1	system us			7195	Connec	Slave has read all rela
2	system us			7196	Connec	Connecting to master

图11-13: mytop实用工具

Jeremy D. Zawodny 编写了 *mytop* 工具 (<http://bit.ly/mytop-clone>)，目前该工具由他和 MySQL 社区共同维护。*mytop* 是开源的，许可证使用 GNU Public License 第 2 版 (GPL v2)。

## InnoTop

404

InnoTop 是另一个系统活动报告，类似于 *top* 命令。受 *mytop* 工具启发，*mytop* 中的很多工具 InnoTop 中也有，不过是专门设计成监控 InnoDB 和 MySQL 服务器的性能的。主要用于监控事务、死锁、外键、查询活动、复制活动、系统变量的主要统计信息及主机的其他详情。

InnoTop 的使用很广泛，被认为是一种常用的性能监控工具。它拥有许多用于动态监控系统功能，如果你主要使用 InnoDB 作为默认（或标准）存储引擎，并需要一个以文本模式运行的良好的监控工具，那就选择 InnoTop。图 11-14 显示了 InnoTop 工具的实例。

cbell@cbell-ubuntu-vm: ~/Downloads/Innotop-1.9.1						
[R0] InnoDB I/O Info (? for help) localhost, 2h9m, InnoDB 10s :-), 5.27						
		I/O Threads				
Thread	Purpose	Thread Status				
0	insert buffer thread	waiting for completed aio requests				
1	log thread	waiting for completed aio requests				
2	read thread	waiting for completed aio requests				
3	read thread	waiting for completed aio requests				
4	read thread	waiting for completed aio requests				
5	read thread	waiting for completed aio requests				
6	write thread	waiting for completed aio requests				
7	write thread	waiting for completed aio requests				
8	write thread	waiting for completed aio requests				
9	write thread	waiting for completed aio requests				
		Pending I/O				
Async Rds	Async Wrt	IBuf Async Rds	Sync I/Os	Log Flushes	Log I/Os	
			0	0	0	0
		File I/O Misc				
OS Reads	OS Writes	OS fsyncs	Reads/Sec	Writes/Sec	Bytes/Sec	
500	37	32	0.00	0.00	0	

图11-14: InnoTop实用工具



InnoTop 工具的许可证使用 GNU Public License 第 2 版 (GPL v2)。

## MONyog

MySQL Monitor and Advisor (MONyog) (<http://www.webyog.com/en>) 是另一个好的 MySQL 监控工具。它可以为安全和性能的主要组件设置参数, 它还包含有助于服务器性能调优的工具。另外, 还可以设置事件以监控特定参数, 并在系统达到指定临界点时发出警告。MONyog 的主要功能是:

- 监控服务器资源
- 识别运行不佳的 SQL 语句
- 监控服务器日志 (如错误日志)
- 实时监控查询性能, 并识别长时间运行的查询
- 预警重大事件

此外, 如果想要用图形输出结果, Monyog 还提供了 GUI 组件。

## MySQL 基准测试套件

基准测试 (benchmarking) 是这样一个过程: 确定系统在某种负载下是如何运行的。基准测试的方法各不相同, 算得上是一门艺术。基准测试的目的是: 分别在服务器处于轻负载、中等负载和高负载的情况下, 运行定义良好的测试实例, 并衡量和记录系统的统计信息。实际上, 基准测试为系统性能设置了期望。

这点很重要, 因为如果系统没有按照期望的那样运行, 它将会给出一些提示。例如, 如果在某段时间内用户反映服务器变慢, 如何知道服务器运行差? 假设已经检查了所有常规信息——内存、磁盘等, 而所有这些都在正常范围内, 没有出现错误和异常。那么这时, 你怎么知道系统是否运行变慢?

使用基准测试。重新运行基准测试实例, 如果执行结果过大 (或者过小, 具体取决于衡量指标), 说明系统的性能没有期望的好。

还可以使用 MySQL 基准测试套件建立自己的基准, 基准测试工具位于 *sql-bench* 文件夹中, 通常被包含在源码发行版中。基准测试是用 Perl 开发的, 使用 Perl DBI 模块访问服务器。如果没有安装 Perl 或 Perl DBI 模块, 请参看在线 MySQL 参考手册中的“在 UNIX 上安装 Perl”一节。

使用以下命令运行基准测试套件:

```
./run-all-tests --server=mysql --cmp=mysql --user=root
```



这个命令将运行一整套标准基准测试，记录当前结果，并将当前结果与以前在 MySQL 服务器上运行的测试结果进行比较。示例 11-4 给出了在资源有限的系统上运行这个命令所输出的部分结果。

示例 11-4: MySQL 基准测试套件的结果

```
cbell@cbell-mini:~/source/bzr/mysql-6.0-review/sql-bench$
Benchmark DBD suite: 2.15
Date of test:      2009-12-01 19:54:19
Running tests on:  Linux 2.6.28-16-generic i686
Comments:
Limits from:      mysql
Server version:    MySQL 6.0.14 alpha debug log
Optimization:      None
Hardware:
alter-table: Total time: 77 wallclock secs
  ( 0.12 usr 0.05 sys + 0.00 cusr 0.00 csys = 0.17 CPU)
ATIS: Total time: 150 wallclock secs
  (20.22 usr 0.56 sys + 0.00 cusr 0.00 csys = 20.78 CPU)
big-tables: Total time: 135 wallclock secs
  (45.73 usr 1.16 sys + 0.00 cusr 0.00 csys = 46.89 CPU)
connect: Total time: 1359 wallclock secs
  (200.70 usr 30.51 sys + 0.00 cusr 0.00 csys = 231.21 CPU)
...
```

尽管这个命令的结果现在还没多大用处，回想一下基准测试是用来跟踪性能随时间的变化。每次运行基准测试套件的时候，都应该将其与已知的基准和上几次基准检查结果相比较。因为负载会影响基准测试，增量进行基准测试能够减轻全天候运行的系统上的负载影响。

例如，如果发现时间突然从一个时间点跳到另一个时间点，说明可能出现了性能下降。还要对比具体值，比如用户和系统时间。当然，如果这些值大多都增加了，说明系统可能负载过重。这时，应该检查进程列表，确定是否有大量用户和查询正在运行。如果是，则在负载较少的时候再次运行基准测试套件，然后对比结果值。如果这些值降低了，可以推断可能是由于负载零散。另一方面，如果值还是偏大（因此系统变慢），就应该开始审查为什么系统比基准测试慢。

## benchmark 函数

MySQL 中有一个内置函数 `benchmark()`，用来执行一个简单表达式，并获得一个基准结果。它的最佳用处是测试其他函数或表达式，以确定它们是否导致延迟。这个函数有两个参数：循环计数器和需要测试的表达式，以下例子显示了 `CONCAT` 函数迭代运行 10 000 000 次的结果：

```
mysql> SELECT BENCHMARK(10000000, "SELECT CONCAT('te','s',' t')");
+-----+
| BENCHMARK(10000000, "SELECT CONCAT('te','s',' t')") |
+-----+
|                                                    0 |
+-----+
1 row in set (0.06 sec)
```

这个函数的输出结果是它运行 `benchmark` 函数所花费的时间。在这个例子中，它花费了 0.06 秒运行迭代。如果研究复杂的查询，则可考虑部分进行测试。你会发现问题仅与查询的某个部分有关。更多内容请参考在线 MySQL 参考手册。

目前我们已经讨论了监控 MySQL 的各种工具，并介绍了一些最佳实践，接下来，我们讨论使用日志文件捕获和保存操作及诊断信息。

## 服务器日志

如果你是一个经验丰富的 Linux 或 UNIX 管理员，那么你应该熟悉日志的概念及其重要性。MySQL 服务器也是在相同的环境中诞生的，因此，MySQL 也有一些日志，包括错误、事件和数据变更相关的重要信息。

本节探讨 MySQL 服务器中的各种日志，包括每种日志在监控和性能改善中的作用。日志文件可以提供大量关于过去事件的信息。

MySQL 服务器中有以下几种类型的日志：

- 常规查询日志
- 慢查询日志
- 错误日志
- 二进制日志

可以使用启动选项开启或关闭任意一种日志。大多数安装的 MySQL 都至少启用了错误日志。

常规查询日志，顾名思义，包含有关服务器行为的信息，如客户端连接，以及发送到服务器的命令副本。可以想象，这个日志增长得很快。每当想要诊断关于客户端的错误或确定某个命令是由哪个客户端发出的，都可以检查常规查询日志。



常规查询日志中的命令的顺序与它们从客户端返回的顺序一致，可能无法真实地反映它们的实际执行顺序。

408

通过设定 `--general-log` 启动选项开启常规查询日志，还可以使用 `--log-output` 启动选项指定日志文件的名称。这些选项都有等价的动态变量。例如，`SET GLOBAL log_output = FILE`，将正在运行的服务器的日志结果写入一个文件中，最后，可以使用 `SHOW VARIABLES` 命令读取到这些变量的值。

慢查询日志保存长时间运行的查询的副本，它与常规日志的格式相同，同样可以通过 `--log-slow-queries` 启动选项来控制慢查询日志。服务器变量 `log_query_time`（以秒为单位）用来控制什么查询被记录到慢查询日志中。应该调整这个变量以满足服务器和应用程序的期望，有助于追踪那些比期望运行慢的查询。使用 `FILE` 或 `TABLE` 选项将日志条目发送到文件或表中，或使用 `BOTH` 选项将日志条目发送到文件和表中。

慢查询日志是个非常有效的工具，它可以在用户抱怨之前追踪出查询问题。当然，我们的目标是保持这个日志很小，最好一直为空。这并不是说要把这个变量设置得很高，相反，应该将它设置为期望值，然后在期望值或环境发生变化时调整这个变量的值。



默认情况下 `slave` 不记录慢查询。但是，如果使用 `--log-slow-slave-statements` 选项，则会把运行慢的事件写入慢日志。

错误日志包含了 MySQL 服务器启动或停止时收集的信息，还包含服务器运行时产生的错误。当开始分析 MySQL 服务器宕机或受损时，应该首先查看错误日志。在有些操作系统上，错误日志还包含了堆栈跟踪（或核心转储）信息。

可以使用 `--log-error` 启动选项开启或关闭错误日志。错误日志的默认名字是主机名加上扩展名 `.err`，默认保存在基目录中（与主机的数据目录位置相同），但可以通过 `general_log_file` 选项设置路径来重写。

如果使用 `--console` 启动服务器，错误会被同时写入标准错误输出和错误日志中。

409 二进制日志存储服务器上数据的所有变更，以及服务器上执行的原始命令的统计信息。

在线 MySQL 参考手册中这样描述：二进制日志是用来做备份的。然而，实践告诉我们，二进制日志更流行的用途是复制。

二进制日志的独特格式使得它可以用于增量备份。通过刷新和轮换（关闭日志然后打开一个新日志）二进制日志，把每次备份时创建的二进制日志文件存储起来，从而保存了自上一次备份以来所产生的一系列变更。使用同样的技术可以执行 PITR，即从备份中恢复数据，然后应用二进制日志到指定点或时间。更多关于二进制日志的信息请参看第 4 章，更多关于 PITR 的信息请参看第 15 章。

因为二进制日志记录每个数据变更的副本，所以它给服务器带来一定的负担，但是相对于它的好处而言，这点小代价是值得的。但是，诸如磁盘配置和存储引擎这样的系统配置，可能会极大地影响二进制日志的开销。例如，如果使用 InnoDB 存储引擎，则不存在并发提交。在使用二进制日志和 InnoDB 处理写频繁的情境时，这可能是个问题。

使用 `--log-bin` 启动项开启二进制日志，并指定二进制日志的根文件名。服务器在文件名的后面附加上一个数字序列，允许自动和手动地执行日志轮换。虽然通常并没有必要，但可以使用 `--logbin-index` 启动选项更改二进制日志的索引名。使用 `FLUSH LOGS` 命令执行日志轮换。

分别使用 `--binlog-do-db` 和 `--binlog-ignore-db` 设定日志中记录什么（包容性日志）或不记录什么（排他性日志）。

## 性能模式

这一节我们讨论性能模式（Performance Schema）功能，用于衡量服务器的内部执行，有助于诊断性能问题。这一节是功能介绍，包含一些简要的入门指南，而没有涵盖全部可能的配置、启动参数和选项，也不是一个使用性能模式视图的完整教程。要了解详情如何配置和使用性能模式表，请参考在线参考手册中的“MySQL 性能模式”一节。

性能模式是最近才加入 MySQL 服务器的功能，表现为名为 `performance_schema`（有时都是大写字母）的数据库。其中包括一组动态表（保存在内存中），能够查看服务器执行的底层指标。这个功能在 5.5.3 版本中引入。性能模式提供了服务器执行的元数据，深入执行的代码行。确实，从某个特定的源文件精确监控某个机制是有可能的。

因此，性能模式通常被认为是一个开发工具，用于诊断服务器代码本身的执行情况。因为大部分时候性能模式都用于诊断死锁、互斥和线程问题。但是，远不止这些呢！还可以使用性能模式获取查询优化的阶段指标、文件 I/O、连接，等等。没错，它是非常底层的，



真的能够让你看到源代码。尽管多数指标面向特定的代码组件，这个工具还能将历史数据和当前值作为指标。如果需要隔离特定的用例来诊断性能难题，这点特别有用。

你可能会想，这会给服务器带来极大的负担，甚至严重影响性能。对某些外部监控方案来说，确实是这样的。但性能模式的设计理念是对服务器的性能产生极少的影响甚至没有影响。因为这个功能与服务器相互纠缠：它利用了服务器的很多优化，而外部工具不能。

下面几个小节将简单介绍性能模式中的一些术语和概念，然后我们介绍如何使用这个功能诊断性能问题。

## 概念

这一节介绍性能模式的基本概念，以便使用性能模式收集指标。

性能模式在数据库列表（SHOW DATABASES）中显示为 performance\_schema，包含大量动态表，通过使用 SHOW TABLES 命令可以查看。示例 11-5 列出了 MySQL 5.6 服务器早期发行版中的表。

表的数量很可能随着以后服务器的发行版而增加。

示例11-5： performance\_schema中的表

```
mysql> SHOW TABLES;;
+-----+
| Tables_in_Performance Schema |
+-----+
| accounts                      |
| cond_instances                |
| events_stages_current         |
| events_stages_history         |
| events_stages_history_long    |
| events_stages_summary_by_account_by_event_name |
| events_stages_summary_by_host_by_event_name |
| events_stages_summary_by_thread_by_event_name |
| events_stages_summary_by_user_by_event_name |
| events_stages_summary_global_by_event_name |
| events_statements_current     |
| events_statements_history     |
| events_statements_history_long |
| events_statements_summary_by_account_by_event_name |
| events_statements_summary_by_digest |
```

```

| events_statements_summary_by_host_by_event_name |
| events_statements_summary_by_thread_by_event_name |
| events_statements_summary_by_user_by_event_name |
| events_statements_summary_global_by_event_name |
| events_waits_current |
| events_waits_history |
| events_waits_history_long |
| events_waits_summary_by_account_by_event_name |
| events_waits_summary_by_host_by_event_name |
| events_waits_summary_by_instance |
| events_waits_summary_by_thread_by_event_name |
| events_waits_summary_by_user_by_event_name |
| events_waits_summary_global_by_event_name |
| file_instances |
| file_summary_by_event_name |
| file_summary_by_instance |
| host_cache |
| hosts |
| mutex_instances |
| objects_summary_global_by_type |
| performance_timers |
| rwlock_instances |
| session_account_connect_attrs |
| session_connect_attrs |
| setup_actors |
| setup_consumers |
| setup_instruments |
| setup_objects |
| setup_timers |
| socket_instances |
| socket_summary_by_event_name |
| socket_summary_by_instance |
| table_io_waits_summary_by_index_usage |
| table_io_waits_summary_by_table |
| table_lock_waits_summary_by_table |
| threads |
| users |
+-----+
52 rows in set (0.01 sec)

```

性能模式监控事件。事件是已经生效（在代码中启用，称为“监控点”）的任意不相关的执行，而且持续时间可测量。例如，事件可以是一个方法调用，互斥锁/解锁，或者文件 I/O。事件存储为当前事件（最近值）、历史值或概要（聚集值）。



性能模式事件与二进制日志事件不同。

因此，监控器是由服务器（源）中的监控点构成的，这些监控点在执行时产生了事件。监控器必须启用才能产生事件。

使用 `setup_actors` 表监控特定用户（线程）。使用 `setup_objects` 表监控特定表或某个数据库中的所有表。目前仅支持表对象。

定时器（timer）是以持续时间测量的一种执行。定时器分为空闲（idle）、等待（wait）、阶段（stage）和语句（statement）。更改定时器的持续时间可以改变测量的频率。值为 `CYCLE`、`NANOSECOND`、`MICROSECOND`、`MILLISECOND` 和 `TICK`。检查 `performance_timers` 表中的行可以查看可用的定时器。

配置表用来启用或禁用行为体（actor）、监控器、对象（表）和定时器。

## 入门

性能模式可以在启动或运行时启用。检查 `performance_schema` 变量，确定服务器是否支持性能模式及其是否开启。如果值为 `ON` 说明启用了性能模式。使用 `--performance-schema` 启动选项在启动时开启性能模式：

```
[mysqld]  
...  
performance_schema=ON  
...
```

在启动时开启性能模式和配置需要监控的事件，需要用到几个启动选项。取决于收集的详细级别，启动时开启性能模式可能会很复杂。幸好全部强制和可选选项及它们的值都保存在配置文件中。如果想要在某种可控条件下收集某个服务器的全部事件，那么在启动时开启性能模式会有所帮助。

但是，大多数管理员都想在运行时开启性能模式。使用 `--performance-schema` 启动变量或者通过配置文件，开启性能模式。一旦开启以后，必须配置需要记录哪些事件，包括修改配置行和配置表。这一小节将介绍开启事件和监控的过程，为诊断性能问题收集数据做准备。

要开启性能模式进行监控，首先要设置定时器，设置事件，并启用想要监控的监控点。

例如，如果要监控所有 `SHOW GRANTS` 事件，首先为这个语句对象设置定时器。这里我们

使用标准的 NANOSECOND 定时。检查 setup\_timers 表查看当前设置：

```
mysql> select * from setup_timers;
+-----+-----+
| NAME      | TIMER_NAME |
+-----+-----+
| idle       | MICROSECOND |
| wait       | CYCLE       |
| stage      | NANOSECOND  |
| statement  | NANOSECOND  |
+-----+-----+
4 rows in set (0.01 sec)
```

接下来，按照以下步骤为这个 SQL 语句开启监控点。这里针对指定命令 (SHOW GRANTS)，我们将 setup\_instruments 表中列的值设置为 YES。具体来说，就是开启指标的监控点和定时器属性：

```
mysql> UPDATE setup_instruments SET enabled='YES', timed='YES'
WHERE name = 'statement/sql/show_grants';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

下一步，开启 events\_statements\_current 和 events\_statements\_history 语句的 consumer：

```
mysql> UPDATE setup_consumers SET enabled='YES'
WHERE name = 'events_statements_current';
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
mysql> UPDATE setup_consumers SET enabled='YES'
WHERE name = 'events_statements_history';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

现在，执行 SHOW GRANTS 命令，检查 events\_statements\_current 和 events\_statements\_history 表：

```
414 mysql> show grants \G
***** 1. row *****
Grants for root@localhost: GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost'
IDENTIFIED BY PASSWORD '*81F5E21E35407D884A6CD4A731AEBFB6AF209E1B'
WITH GRANT OPTION
***** 2. row *****
Grants for root@localhost: GRANT PROXY ON '' TO 'root'@'localhost' WITH
```



```
GRANT OPTION
2 rows in set (0.01 sec)
```

```
mysql> select * from events_statements_current \G
```

```
***** 1. row *****
      THREAD_ID: 22
      EVENT_ID: 80
      END_EVENT_ID: NULL
      EVENT_NAME: statement/sql/select
      SOURCE: mysqld.cc:903
      TIMER_START: 13104624563678000
      TIMER_END: NULL
      TIMER_WAIT: NULL
      LOCK_TIME: 136000000
      SQL_TEXT: select * from events_statements_current
      DIGEST: NULL
      DIGEST_TEXT: NULL
      CURRENT_SCHEMA: performance_schema
      OBJECT_TYPE: NULL
      OBJECT_SCHEMA: NULL
      OBJECT_NAME: NULL
      OBJECT_INSTANCE_BEGIN: NULL
      MYSQL_ERRNO: 0
      RETURNED_SQLSTATE: NULL
      MESSAGE_TEXT: NULL
      ERRORS: 0
      WARNINGS: 0
      ROWS_AFFECTED: 0
      ROWS_SENT: 0
      ROWS_EXAMINED: 0
      CREATED_TMP_DISK_TABLES: 0
      CREATED_TMP_TABLES: 0
      SELECT_FULL_JOIN: 0
      SELECT_FULL_RANGE_JOIN: 0
      SELECT_RANGE: 0
      SELECT_RANGE_CHECK: 0
      SELECT_SCAN: 1
      SORT_MERGE_PASSES: 0
      SORT_RANGE: 0
      SORT_ROWS: 0
      SORT_SCAN: 0
      NO_INDEX_USED: 1
      NO_GOOD_INDEX_USED: 0
      NESTING_EVENT_ID: NULL
      NESTING_EVENT_TYPE: NULL
```

```
mysql> select * from events_statements_history \G
```

```
***** 1. row *****
```

```

      THREAD_ID: 22
      EVENT_ID: 77
    END_EVENT_ID: 77
      EVENT_NAME: statement/sql/select
        SOURCE: mysqld.cc:903
      TIMER_START: 12919040536455000
      TIMER_END: 12919040870255000
      TIMER_WAIT: 333800000
      LOCK_TIME: 143000000
      SQL_TEXT: select * from events_statements_history
      DIGEST: 77d3399ea8360ffc7b8d584c0fac948a
      DIGEST_TEXT: SELECT * FROM `events_statements_history`
    CURRENT_SCHEMA: performance_schema
      OBJECT_TYPE: NULL
      OBJECT_SCHEMA: NULL
      OBJECT_NAME: NULL
    OBJECT_INSTANCE_BEGIN: NULL
      MYSQL_ERRNO: 0
    RETURNED_SQLSTATE: NULL
      MESSAGE_TEXT: NULL
      ERRORS: 0
      WARNINGS: 0
    ROWS_AFFECTED: 0
      ROWS_SENT: 1
      ROWS_EXAMINED: 1
    CREATED_TMP_DISK_TABLES: 0
    CREATED_TMP_TABLES: 0
      SELECT_FULL_JOIN: 0
    SELECT_FULL_RANGE_JOIN: 0
      SELECT_RANGE: 0
    SELECT_RANGE_CHECK: 0
      SELECT_SCAN: 1
    SORT_MERGE_PASSES: 0
      SORT_RANGE: 0
      SORT_ROWS: 0
      SORT_SCAN: 0
      NO_INDEX_USED: 1
    NO_GOOD_INDEX_USED: 0
      NESTING_EVENT_ID: NULL
    NESTING_EVENT_TYPE: NULL

```

```
***** 2. row *****
```

```

        THREAD_ID: 22
        EVENT_ID: 78
    END_EVENT_ID: 78
        EVENT_NAME: statement/sql/show_grants
        SOURCE: mysqld.cc:903
    TIMER_START: 12922392541028000
    TIMER_END: 12922392657515000
    TIMER_WAIT: 116487000
    LOCK_TIME: 0
    SQL_TEXT: show grants
    DIGEST: 63ca75101f4bfc9925082c9a8b06503b
    DIGEST_TEXT: SHOW GRANTS
    CURRENT_SCHEMA: performance_schema
    OBJECT_TYPE: NULL
    OBJECT_SCHEMA: NULL
    OBJECT_NAME: NULL
    OBJECT_INSTANCE_BEGIN: NULL
    MYSQL_ERRNO: 0
    RETURNED_SQLSTATE: NULL
    MESSAGE_TEXT: NULL
    ERRORS: 0
    WARNINGS: 0
    ROWS_AFFECTED: 0
    ROWS_SENT: 0
    ROWS_EXAMINED: 0
    CREATED_TMP_DISK_TABLES: 0
    CREATED_TMP_TABLES: 0
    SELECT_FULL_JOIN: 0
    SELECT_FULL_RANGE_JOIN: 0
    SELECT_RANGE: 0
    SELECT_RANGE_CHECK: 0
    SELECT_SCAN: 0
    SORT_MERGE_PASSES: 0
    SORT_RANGE: 0
    SORT_ROWS: 0
    SORT_SCAN: 0
    NO_INDEX_USED: 0
    NO_GOOD_INDEX_USED: 0
    NESTING_EVENT_ID: NULL
    NESTING_EVENT_TYPE: NULL

```

\*\*\*\*\* 3. row \*\*\*\*\*

```

        THREAD_ID: 22
        EVENT_ID: 74
    END_EVENT_ID: 74
        EVENT_NAME: statement/sql/show_grants

```

```

SOURCE: mysqld.cc:903
TIMER_START: 12887992696398000
TIMER_END: 12887992796352000
TIMER_WAIT: 99954000
LOCK_TIME: 0
SQL_TEXT: show grants
DIGEST: 63ca75101f4bfc9925082c9a8b06503b
DIGEST_TEXT: SHOW GRANTS
CURRENT_SCHEMA: performance_schema
OBJECT_TYPE: NULL
OBJECT_SCHEMA: NULL
OBJECT_NAME: NULL
OBJECT_INSTANCE_BEGIN: NULL
MYSQL_ERRNO: 0
417 RETURNED_SQLSTATE: NULL
MESSAGE_TEXT: NULL
ERRORS: 0
WARNINGS: 0
ROWS_AFFECTED: 0
ROWS_SENT: 0
ROWS_EXAMINED: 0
CREATED_TMP_DISK_TABLES: 0
CREATED_TMP_TABLES: 0
SELECT_FULL_JOIN: 0
SELECT_FULL_RANGE_JOIN: 0
SELECT_RANGE: 0
SELECT_RANGE_CHECK: 0
SELECT_SCAN: 0
SORT_MERGE_PASSES: 0
SORT_RANGE: 0
SORT_ROWS: 0
SORT_SCAN: 0
NO_INDEX_USED: 0
NO_GOOD_INDEX_USED: 0
NESTING_EVENT_ID: NULL
NESTING_EVENT_TYPE: NULL
***** 4. row *****
THREAD_ID: 22
EVENT_ID: 75
END_EVENT_ID: 75
EVENT_NAME: statement/sql/select
SOURCE: mysqld.cc:903
TIMER_START: 12890520653158000
TIMER_END: 12890521011318000
TIMER_WAIT: 358160000

```



```

LOCK_TIME: 148000000
SQL_TEXT: select * from events_statements_current
DIGEST: f06ce227c4519dd9d9604a3f1cfe3ad9
DIGEST_TEXT: SELECT * FROM `events_statements_current`
CURRENT_SCHEMA: performance_schema
OBJECT_TYPE: NULL
OBJECT_SCHEMA: NULL
OBJECT_NAME: NULL
OBJECT_INSTANCE_BEGIN: NULL
MYSQL_ERRNO: 0
RETURNED_SQLSTATE: NULL
MESSAGE_TEXT: NULL
ERRORS: 0
WARNINGS: 0
ROWS_AFFECTED: 0
ROWS_SENT: 1
ROWS_EXAMINED: 1
CREATED_TMP_DISK_TABLES: 0
CREATED_TMP_TABLES: 0
SELECT_FULL_JOIN: 0
SELECT_FULL_RANGE_JOIN: 0
SELECT_RANGE: 0
SELECT_RANGE_CHECK: 0
SELECT_SCAN: 1
SORT_MERGE_PASSES: 0
SORT_RANGE: 0
SORT_ROWS: 0
SORT_SCAN: 0
NO_INDEX_USED: 1
NO_GOOD_INDEX_USED: 0
NESTING_EVENT_ID: NULL
NESTING_EVENT_TYPE: NULL

```

\*\*\*\*\* 5. row \*\*\*\*\*

```

THREAD_ID: 22
EVENT_ID: 76
END_EVENT_ID: 76
EVENT_NAME: statement/sql/select
SOURCE: mysqld.cc:903
TIMER_START: 12895480384972000
TIMER_END: 12895480736605000
TIMER_WAIT: 351633000
LOCK_TIME: 144000000
SQL_TEXT: select * from events_statements_history
DIGEST: 77d3399ea8360ffc7b8d584c0fac948a
DIGEST_TEXT: SELECT * FROM `events_statements_history`

```

```

CURRENT_SCHEMA: performance_schema
OBJECT_TYPE: NULL
OBJECT_SCHEMA: NULL
OBJECT_NAME: NULL
OBJECT_INSTANCE_BEGIN: NULL
MYSQL_ERRNO: 0
RETURNED_SQLSTATE: NULL
MESSAGE_TEXT: NULL
ERRORS: 0
WARNINGS: 0
ROWS_AFFECTED: 0
ROWS_SENT: 1
ROWS_EXAMINED: 1
CREATED_TMP_DISK_TABLES: 0
CREATED_TMP_TABLES: 0
SELECT_FULL_JOIN: 0
SELECT_FULL_RANGE_JOIN: 0
SELECT_RANGE: 0
SELECT_RANGE_CHECK: 0
SELECT_SCAN: 1
SORT_MERGE_PASSES: 0
SORT_RANGE: 0
SORT_ROWS: 0
SORT_SCAN: 0
NO_INDEX_USED: 1
NO_GOOD_INDEX_USED: 0
NESTING_EVENT_ID: NULL
NESTING_EVENT_TYPE: NULL

```

5 rows in set (0.00 sec)

注意，events\_statements\_table的输出结果仅仅是最后一次记录的执行语句，而events\_statements\_history的输出结果是所有开启的事件上的最近查询。在这个例子中，我们同时开启了statement/sql/select和statement/sql/show\_grants的监控点，所以这两种类型的事件都会显示。

虽然这个例子很简单，不过我们可以从中获取很多信息。例如，我们看到输出结果中有定时信息，比如查询的开始时间和结束时间，还有锁的时间等。我们还看到了警告和错误计数，查询优化信息，以及是否使用索引等信息。

这个例子中的步骤代表了我们的开启监控点和事件的步骤。总结起来，使用性能模式进行监控应该遵守以下步骤：

1. 设置定时器（适用于有定时元素的监控点）

## 2. 开启监控点

## 3. 开启 consumer

### 过滤事件

过滤事件有两种技术：事前过滤（prefiltering）和事后过滤（postfiltering）。

事前过滤的实现方式是：修改性能模式的配置，仅开启从某些生产者（producer）到某些消费者（consumer）所收集的事件。事前过滤减少了开销，避免了向历史表填充不必要的指标。事前过滤的缺点是需要测试之前预测想要检查哪些事件。

事后过滤的实现方式通常是：开启生产者和消费者，收集足够多的信息。在性能模式表中使用 WHERE 从句收集数据之后执行过滤。事后过滤是基于单个用户的（在 WHERE 从句中）。当你不确定需要收集哪些事件的时候，就使用事后过滤。例如，当没有可重复的用例时。

选择哪种过滤取决于需要收集的数据量。如果知道自己要什么（需要测量的指标等），而且只需要记录这些事件，那么就选择事前过滤。另一方面，如果不知道要什么，或者需要随时间产生数据指标，则可能需要考虑事后过滤，使用 SELECT 语句检查性能模式表缩小查找范围。

420

## 使用性能模式诊断性能问题

这一节介绍使用性能模式诊断问题的另一种方法，包括保证服务器回到原始状态的优化方法。

跟参考手册中的例子一样，这个方法假定有一组操作在多个数据库之间出现了重复性问题。

注意：用例不是一成不变的，可能需要重现的不只是数据和查询。例如，如果诊断的问题与负载或某些条件（大量连接、某个应用程序，等等）有关，可能需要重现负载和类似的条件。

在使用性能模式之前应该解决的另一个条件是需要使用什么参数、变量和选项来调整服务器。如果不知道哪些需要调整就胡乱摆弄服务器，一点用都没有。也许你并不明确地知道究竟哪些应该调整，但这时你应该有主意了。而且，确保做出更改之前记下当前值。通常，在调整的过程中，一次只更改一个参数或选项，然后对比更改前后的性能变化。如果性能没有明显变化，就应该在更改下一个参数或选项之前恢复到原来的值。

使用性能模式诊断性能问题的步骤如下：

1. 查询 `setup_instruments` 表，找到所有相关的监控点，然后开启它们。
2. 设置定时器，即记录的频率。大多数时候默认调整后的定时值。如果更改了定时器，则记录它们的初始值。
3. 找到与监控点相关的消费者（consumers，事件表），然后开启它们。一定要开启 `current`、`history` 和 `history_long` 变量。
4. 截断（truncate）`*history` 和 `*history_long` 表，确保开始状态是“干净的”。
5. 重现问题。
6. 查询性能模式表。如果服务器上运行了多个客户端，使用线程 ID 隔离行。
7. 观察值并记录它们。
8. 调整一个选项 / 参数 / 变量集。
9. 返回步骤 5 重复这个过程，直到性能得到提升。
10. 截断 `*history` 和 `*history_long` 表，确保结束状态是“干净的”。
11. 禁用之前开启的事件。
12. 禁用之前开启的监控点。
13. 将定时器返回到原始状态。
14. 再一次截断 `*history` 和 `*history_long` 表，确保结束状态是“干净的”。

## MySQL 的监控分类

前面几节介绍了很多监控 MySQL 的方法。有些方法，比如系统变量和状态变量，需要检查很多指标，以发现性能、访问或资源上的问题原因。了解可以或应该使用什么，对解决问题来说是非常重要的，而且能节省很多时间。

我们需要一个不同设备、工具和指标的映射以监控 MySQL。下面这个表列出了有效监控 MySQL 服务器需要用到的监控设备的分类。表 11-1 按照关注点、设备和指标对任务进行了分类，还给出了相应的例子。



表11-1: MySQL监控分类

关注点	设备	指标	示例
性能	系统变量	查询缓存	SHOW VARIABLES LIKE '%query_cache%'
性能	状态变量	插入的数量	SHOW STATUS LIKE 'com_insert'
性能	状态变量	删除的数量	SHOW STATUS LIKE 'com_delete'
性能	状态变量	表锁冲突	SHOW STATUS LIKE 'table_locks_waited'
性能	登录	慢查询	SELECT * FROM slow_log ORDER BY query_time DESC
性能	登录	一般	SELECT * FROM general_log
性能	登录	错误	--log-error=file name (startup variable)
性能	性能模式	线程信息	SELECT * FROM threads
性能	性能模式	互斥体信息	SELECT * FROM events_wait_current
性能	性能模式	互斥体信息	SELECT * FROM mutex_instances
性能	性能模式	文件使用小结	SELECT * FROM file_summary_by_instance
性能	存储引擎功能	InnoDB 状态	SHOW ENGINE innodb STATUS
性能	存储引擎功能	InnoDB 统计数据	SHOW STATUS LIKE '%Innodb%'
性能	外部工具	进程列表	mysqladmin -uroot --password process list --sleep 3
性能	外部工具	连接情况 (图)	MySQL Workbench
性能	外部工具	内存情况 (图)	MySQL Workbench
性能	外部工具	InnoDB 读取的行	MySQL Workbench
性能	外部工具	日志	MySQL Workbench
性能	外部工具	全部变量	MySQL Workbench
性能	外部工具	查询计划执行 <sup>a</sup>	MySQL Workbench
性能	外部工具	基准测试	MySQL Benchmark Suite
可用性	状态变量	连接的线程	SHOW STATUS LIKE 'threads_connected'
可用性	操作系统工具	可访问性	ping
可用性	外部工具	可访问性	mysqladmin -uroot --password extended-status --relative --sleep 3
资源	状态变量	支持的存储引擎	SHOW ENGINES
资源	操作系统工具	CPU 使用情况	top -n 1 -pid mysqld_pid
资源	操作系统工具	RAM 使用情况	top -n 1 -pid mysqld_pid
资源	MySQL 实用工具	磁盘使用情况	mysqldiskusage
资源	MySQL 实用工具	服务器信息	mysqlserverinfo
资源	MySQL 实用工具	复制情况	mysqlrepadmin

<sup>a</sup> 可以使用 EXPLAIN SQL 命令。

我们发现,大量监控技术都是面向性能监控的。这并不奇怪,因为数据库服务器通常是很多应用程序的焦点,而且可能有成千上万个用户。从这个表中我们还发现,诊断性能

问题可以使用多个设备。通常这些设备和指标能够引导你找到解决性能问题的方法。既然我们已经有了如何监控 MySQL 的路径图，这可以帮助我们重点关注某些适当的设备。

通常情况下，需要审查特定数据库（或多个数据库）的性能问题，或者需要提升那些造成应用程序性能瓶颈的查询的性能。下面的章节我们将研究提升数据库和查询性能的技术和最佳实践。

423

## 数据库性能

监控单个数据库的性能是 MySQL 的功能之一，MySQL 社区和第三方开发人员已经改善了 MySQL 这些方面的功能。MySQL 包含一些改善性能的基本工具，但它们不像其他系统优化工具那样成熟。由于这些限制，大多数 MySQL DBA 通过关系查询优化技术方面的经验获益。我们发现有一些非常棒的参考资料详细介绍了数据库性能，很多读者可能已经熟悉了数据库优化的基本知识。这里罗列了一些资料供大家参考：

- *Refactoring SQL Applications*，作者是 Stephane Faroult 和 Pascal L’Hermite (O’Reilly)
- *SQL and Relational Theory: How to Write Accurate SQL Code*，作者是 C.J.Date (O’Reilly)
- *SQL Cookbook*，作者是 Anthony Mollinaro (O’Reilly)

我们主要讨论如何使用 MySQL 中的工具优化数据库，而不再重新介绍查询优化技术。我们将使用一个简单的例子和一个数据库样例来说明如何使用 MySQL 中的查询性能命令。下一节将介绍改善数据库性能的最佳实践。

## 衡量数据库的性能

一般的数据库管理系统都提供了分析工具和索引工具，这些工具产生一些统计信息以优化索引。虽然有一些基本元素可以帮助改善 MySQL 数据库的性能，但是没有开源的高级分析工具可用。

虽然基本的 MySQL 安装包不包含监控数据库改善的正式工具，但是 MySQL 企业版管理器套件提供了大量性能监控功能，我们将在第 16 章中更详细地讨论这个工具。

幸运的是，MySQL 提供了一些简单的工具，有助于确定表和查询是否是最优的。它们都是 SQL 命令，包括 EXPLAIN、ANALYZE TABLE 和 OPTIMIZE TABLE。以下章节将详细介绍这些命令。

### 使用 EXPLAIN

EXPLAIN 命令给出如何执行 SELECT 语句（EXPLAIN 仅对 SELECT 语句有效）的信息。

EXPLAIN 的语法如下所示。请注意，EXPLAIN 与其他数据库系统中的 DESCRIBE 命令类似。

```
[EXPLAIN | DESCRIBE] [EXTENDED] SELECT select options
```

还可以使用 EXPLAIN 和 DESCRIBE 命令查看表的列或分区信息，语法如下：

```
[EXPLAIN | DESCRIBE] [PARTITIONS SELECT * FROM] table_name
```



EXPLAIN *table\_name* 命令的同义命令是 SHOW COLUMNS FROM *table\_name*。

首先讨论 EXPLAIN 命令的第一种用法，即检查 SELECT 命令以查看 MySQL 优化器是如何执行这个语句的。其结果是优化器预计执行该语句需要的 JOIN 操作列表。

这个命令的最佳用处是确定表是否有最佳索引，从而更精确地定位候选行。还可以使用这个结果测试各种优化器的重载选项。这是一个高级技术，一般情况下不提倡使用，但是有些时候你可能遇到这样的查询：采用某些优化器选项查询运行得更快。稍后我们看一个这样的例子。

现在，让我们看看 EXPLAIN 命令的一些实例。下面的例子是在 MySQL 开发和测试时使用的 *sakila* 样本数据库上执行的查询 (<http://bit.ly/sakila-sd>)。

我们首先看一个简单而且看起来没有什么危害的查询。比方说，希望看到所有比 PG 等级高的电影。结果集中的每一行包含以下字段：

id

语句的执行序列号。

select\_type

被执行的语句类型。

table

在这一步操作的表。

type

使用的 JOIN 类型。

possible\_keys

如果有包含主键的索引，可用字段的列表。

key

被优化器选中的主键。

425 key\_len

主键或部分主键的长度。

ref

约束或需要对比的字段。

rows

估计要处理的行数。

extra

优化器的额外信息。



如果 type 字段为 ALL，其是在做全表扫描。应该尽量避免这个，方法是添加索引或重写查询。类似的，如果 type 字段为 INDEX，则执行全索引扫描，这是非常低效的。参见在线 MySQL 参考手册可获取更多 JOIN 类型及其意义。

示例 11-6 显示了 MySQL 优化器是如何执行这个语句的。为了清晰显示，我们使用 \G 做了垂直显示。

这个例子中的表中有一个字段（列）定义为枚举类型。枚举类型接受一组值的列表。如果不使用枚举类型而是定义一个查找表的话，就必须执行 JOIN 来选择指定值的结果。因此，枚举值能够代替小型查找表，所以枚举值可以提升性能。

这是因为枚举值的文本仅保存一次——在表头结构中，行中保存的是数字值，形成一个枚举值的索引（数组索引）。枚举值列表能够节省空间，使得遍历数据更加有效。一个枚举字段类型只能接受一个值。

在下面的例子中，sakila 数据库中的电影表有一个名为 rating 的枚举字段，其值为 G、PG、PG-13、R 和 NC-17。在下面的例子中我们会看到这个枚举值字段是如何在查询中使用（和误用）的。

示例 11-6：一个简单的 SELECT 语句

```
mysql> EXPLAIN SELECT * FROM film WHERE rating > 'PG' \G
***** 1. row *****
    id: 1
  select_type: SIMPLE
    table: film
```



```

      type: ALL
possible_keys: NULL
      key: NULL
     key_len: NULL
        ref: NULL
       rows: 892
  Extra: Using where
1 row in set (0.01 sec)

```

从这个输出结果可以看到优化器只执行了一步，而且没有使用任何索引。这是讲得通的，因为没有使用任何索引列。此外，即使这里包含了一个 WHERE 子句，但是优化器仍然需要做全表扫描。当你考虑使用的列缺少索引时，这可能是正确的选择。然而，如果这个查询运行几百上千次，全表扫描将很浪费时间。在这种情况下，从结果可以看出添加索引可以改善执行情况（参见示例 11-7）。

示例11-7：添加索引优化查询性能

```

mysql> ALTER TABLE film ADD INDEX film_rating (rating);
Query OK, 0 rows affected (0.42 sec)
Records: 0 Duplicates: 0 Warnings: 0

```

给表添加一个索引，然后再测试。示例 11-8 显示了改善的查询计划。

示例11-8：改善的查询计划

```

mysql> EXPLAIN SELECT * FROM film WHERE rating > 'PG' \G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: film_rating
          key: NULL
     key_len: NULL
        ref: NULL
       rows: 892
  Extra: Using where
1 row in set (0.00 sec)

```



有些敏锐的人已经发现了问题，忍耐一下，我们马上解决这个问题。

这里，我们看到查询识别到了一个索引（possible\_keys），但是仍然没有使用索引，因

为 key 字段是 NULL。那么我们要做什么呢？对于这个简单的例子，你可能会注意到预计读取的只有 892 行，而实际有 1000 行，结果集中只有 418 行。显然，如果这个查询只读取 42% 的行，它将执行得更快。

现在看看通过使用 EXTENDED 关键字是否可以从优化器中获得一些额外的信息。这个关键字可以通过 SHOW WARNINGS 命令显示一些额外的信息。应该在 EXPLAIN 命令后面立即执行这个命令。警告文本说明了优化器是如何识别语句中的表和列名、如何在内部重写这个查询、如何应用优化规则的，以及其他关于执行的信息说明。示例 11-9 显示了使用 EXTENDED 关键字的结果。

示例11-9：使用EXTENDED关键字获取更多信息

```
mysql> EXPLAIN EXTENDED SELECT * FROM film WHERE rating > 'PG' \G
```

```
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: film_rating
         key: NULL
        key_len: NULL
         ref: NULL
         rows: 892
    filtered: 100.00
      Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

```
mysql> SHOW WARNINGS \G
```

```
***** 1. row *****
      Level: Note
       Code: 1003
Message: select `sakila`.`film`.`film_id` AS `film_id`,
`sakila`.`film`.`title` AS `title`,`sakila`.`film`.`description` AS `description`,
`sakila`.`film`.`release_year` AS `release_year`,
`sakila`.`film`.`language_id` AS `language_id`,
`sakila`.`film`.`original_language_id` AS `original_language_id`,
`sakila`.`film`.`rental_duration` AS `rental_duration`,
`sakila`.`film`.`rental_rate` AS `rental_rate`,
`sakila`.`film`.`length` AS `length`,
`sakila`.`film`.`replacement_cost` AS `replacement_cost`,
`sakila`.`film`.`rating` AS `rating`,
`sakila`.`film`.`special_features` AS `special_features`,
`sakila`.`film`.`last_update` AS `last_update`
from `sakila`.`film` where (`sakila`.`film`.`rating` > 'PG')
1 row in set (0.00 sec)
```

这次有一个关于优化器的警告，显示了查询的重写形式，包含所有的字段，并显式引用了 WHERE 从句中的列。虽然这说明查询可以被更好地重写，但却没有任何性能上的提升，幸好我们让它更有效了。

如果查询某个特定的 rating 而不是使用范围查询时，让我们看看会发生什么。我们将看到使用和不使用索引的优化。示例 11-10 显示了结果。

#### 示例 11-10：去掉范围查询

```
mysql> EXPLAIN SELECT * FROM film WHERE rating = 'R' \G
```

```
***** 1. row *****
```

```
id: 1
select_type: SIMPLE
table: film
type: ref
possible_keys: film_rating
key: film_rating
key_len: 2
ref: const
rows: 195
Extra: Using where
1 row in set (0.00 sec)
```

```
mysql> ALTER TABLE film DROP INDEX film_rating;
```

```
Query OK, 0 rows affected (0.37 sec)
```

```
Records: 0 Duplicates: 0 Warnings: 0
```

```
mysql> EXPLAIN SELECT * FROM film WHERE rating = 'R' \G
```

```
***** 1. row *****
```

```
id: 1
select_type: SIMPLE
table: film
type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: 892
Extra: Using where
1 row in set (0.00 sec)
```

现在看到有一些小小的改善。注意，第一个查询计划确实使用了索引，而且带来了较大的改善。那么问题来了，为什么优化器不使用索引呢？这里，我们在枚举字段上使用了非唯一索引。这听起来像是个好办法，但实际对于枚举值上的范围查询却没有多大帮助。

然而，我们可以重写上面的查询（实际上有多种方法），以获得更好的性能。让我们再来看看这个查询。

我们希望所有电影的等级比 PG 高。假定对 rating 进行了排序，而且枚举字段反映了顺序。因此，如果枚举索引的每个值与这个顺序对应（如 G=1，PG=2 等），那么看上去维持了原有的顺序，但是如果这个顺序不正确或（像这个例子一样）值列表不完整呢？

在这个例子中，我们希望所有影片的 rating 比 PG 高，从 rating 列表中我们知道这些影片的 rating 是 R 或 NC-17。在不使用范围查询的前提下，如果列举了所有这些值，让我们看看优化器做了些什么。

回想一下，我们删除了索引，所以我们首先试试没有索引的查询，然后添加索引，再看看查询是否有所改善。示例 11-11 显示了改进后的查询。

#### 示例 11-11：改进后的非范围查询

```
mysql> EXPLAIN SELECT * FROM film WHERE rating = 'R' OR rating = 'NC-17' \G
```

```
***** 1. row *****
```

```
id: 1
select_type: SIMPLE
table: film
type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: 892
Extra: Using where
1 row in set (0.00 sec)
```

```
mysql> ALTER TABLE film ADD INDEX film_rating (rating);
```

```
Query OK, 0 rows affected (0.40 sec)
```

```
Records: 0 Duplicates: 0 Warnings: 0
```

```
mysql> EXPLAIN SELECT * FROM film WHERE rating = 'R' OR rating = 'NC-17' \G
```

```
***** 1. row *****
```

```
id: 1
select_type: SIMPLE
table: film
type: ALL
possible_keys: film_rating
key: NULL
key_len: NULL
ref: NULL
rows: 892
```



```
Extra: Using where
1 row in set (0.00 sec)
```

唉，还是不行。我们同样选择了查询有索引的列，但是优化器却不能使用这个索引。但是，优化器做简单的等值比较的时候可以使用索引，因为需要将比较的值保存在索引中。重写这个查询，使用 union 合并两个查询。示例 11-12 显示了重写后的查询。

◀ 430

示例11-12：使用UNION重写查询

```
mysql> EXPLAIN SELECT * FROM film WHERE rating = 'R' UNION
SELECT * FROM film WHERE rating = 'NC-17' \G
```

```
***** 1. row *****
```

```
id: 1
select_type: PRIMARY
table: film
type: ref
possible_keys: film_rating
key: film_rating
key_len: 2
ref: const
rows: 195
Extra: Using where
```

```
***** 2. row *****
```

```
id: 2
select_type: UNION
table: film
type: ref
possible_keys: film_rating
key: film_rating
key_len: 2
ref: const
rows: 210
Extra: Using where
```

```
***** 3. row *****
```

```
id: NULL
select_type: UNION RESULT
table: <union1,2>
type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: NULL
Extra:
```

```
3 rows in set (0.00 sec)
```

成功了！我们看到这个查询计划使用了索引，处理了更少的行。从 EXPLAIN 命令的结果可以看到优化器单独运行每个查询（从第一行到第  $n$  行分步执行），然后在最后一步合并结果。

431



MySQL 有一个会话状态变量 `last_query_cost`，这个变量存储最近一次执行的查询的成本。使用这个变量可比较同一个查询的两个查询计划。例如，执行完每个 EXPLAIN 后，检查这个变量的值，拥有最低成本值的查询被认为是更加高效（耗时更少）的查询。如果该值为 0，则表明没有查询提交编译。

虽然看上去这个例子需要做很多事情却收获很小，但是应该这样考虑：应用程序中有很多这样的查询正在执行，却没有人注意到它的低效率。通常只有在行数大到足够引起注意的时候才会遇到这样类型的查询。在 *sakila* 数据库中，只有 1000 行，但是如果有一百万或数千万行呢？

EXPLAIN 是标准 MySQL 发行版中唯一一个用来分析查询的工具。在线 MySQL 文档中的“优化”一章中介绍了大量技巧来帮助有经验的 DBA 提高各种查询的性能。

## 使用 ANALYZE TABLE

与大多数传统优化器相同，MySQL 优化器使用表的统计信息分析最优查询执行计划。这些统计信息涉及多项信息，包括索引、数值分布和表结构等。

ANALYZE TABLE 命令重新计算一个或多个表的主键分布。这个信息确定 JOIN 操作中表的顺序。ANALYZE TABLE 命令的语法如下所示：

```
ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE table_list
```

每当表上有重大变更（如批量加载数据）时，都应该执行这个命令。在执行的时候，系统必须在这个表上加上读锁。

只能为 MyISAM 和 InnoDB 表更新主键分布。其他存储引擎并不支持这个工具，但如果存储引擎支持索引则所有存储引擎必须将索引的基数统计信息报告给优化器。有些存储引擎，特别是第三方引擎，有它们特有的内置统计信息。示例 11-13 给出了这个命令的典型执行。在没有索引的表上运行这个命令没有任何效果，但也不会导致错误发生。

### 示例11-13: 分析表, 更新主键分布

```
mysql> ANALYZE TABLE film;
```

```
+-----+-----+-----+-----+
| Table      | Op      | Msg_type | Msg_text |
+-----+-----+-----+-----+
| sakila.film | analyze | status   | OK       |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```



如果使用的是 InnoDB 表, 有些时候不应该使用这个命令。参见在线参考手册中的 `innodb_stats_persistent` 获取更多详细内容。

在这个例子中, 我们看到分析完成了, 并且没有异常出现。在这个命令执行的过程中如果出现异常事件, `Msg_type` 字段将为 `info`、`Error` 或 `warning`。在这些情况下, `Msg_text` 字段将显示事件的额外信息。如果你得到的结果不是 `status` 或 `OK`, 应该审查一下。

例如, 如果表的 `.frm` 文件损坏或丢失, 可以查看以下消息。在其他情况下, 输出结果可能表明表是不可读的 (比如权限和访问问题)。此外, 命令执行了存储引擎相关的检查。对 InnoDB 来说, 这个检查更加仔细, 而且如果有错误的话, 可能会看到 InnoDB 相关的错误, 参见示例 11-14。

### 示例11-14: 分析表错误

```
mysql> ANALYZE TABLE test.t1;
```

```
+-----+-----+-----+-----+
| Table  | Op      | Msg_type | Msg_text |
+-----+-----+-----+-----+
| test.t1 | analyze | Error    | Table 'test.t1' doesn't exist |
| test.t1 | analyze | status   | Operation failed |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

使用 `SHOW INDEX` 命令可查看索引的状态。示例 11-15 显示了 `film` 表的输出结果样本。在这个例子中, 我们对每个索引的基数感兴趣, 它是索引中的唯一值的个数估计。为方便起见, 我们在显示结果中省略了其他列。请参看在线 MySQL 参考手册以获取更多关于 `SHOW INDEX` 的信息。

### 示例11-15: film表的索引

```
mysql> SHOW INDEX FROM film \G
```

```
***** 1. row *****
Table: film
```

```

Non_unique: 0
Key_name: PRIMARY
Seq_in_index: 1
Column_name: film_id
Collation: A
Cardinality: 1028
...
***** 2. row *****
      Table: film
      Non_unique: 1
      Key_name: idx_title
      Seq_in_index: 1
      Column_name: title
      Collation: A
      Cardinality: 1028
      ...
***** 3. row *****
      Table: film
      Non_unique: 1
      Key_name: idx_fk_language_id
      Seq_in_index: 1
      Column_name: language_id
      Collation: A
      Cardinality: 2
      ...
***** 4. row *****
      Table: film
      Non_unique: 1
      Key_name: idx_fk_original_language_id
      Seq_in_index: 1
      Column_name: original_language_id
      Collation: A
      Cardinality: 2
      ...
***** 5. row *****
      Table: film
      Non_unique: 1
      Key_name: film_rating
      Seq_in_index: 1
      Column_name: rating
      Collation: A
      Cardinality: 11
      Sub_part: NULL
      Packed: NULL
      Null: YES

```



```
Index_type: BTREE
Comment:
5 rows in set (0.00 sec)
```

## 使用 OPTIMIZE TABLE

434

频繁使用新数据或删除操作，表将很快变得支离破碎，而且根据使用的存储引擎不同，将会出现不同程度的闲置空间或不理想的存储结构。支离破碎的表会导致性能下降，尤其是在表扫描的时候。

OPTIMIZE TABLE 命令可以重构一个或多个表的数据结构。对于长度可变的字段(行)而言，这个命令尤其有用，OPTIMIZE TABLE 命令的语法如下：

```
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE table_list
```



LOCAL 或 NO\_WRITE\_TO\_BINLOG 关键字防止命令被写入二进制日志中（因此也不会在复制拓扑中被复制）。如果希望在复制数据时进行实验或调整，或者如果希望从二进制日志中省略这一步，而且在 PITR 时不再重放，那么这个命令很有用。

每次表上有重大变更（例如大量的删除和插入）时，都应该运行这个命令。这个操作将数据元素重组到更优的结构中，而且可能会运行很长时间（在表上持有写锁）。所以，这个操作最好在低负载的时候运行。

如果表不能被重新组织（例如如果不存在可变长度的记录或不存在碎片），这个命令将重新创建表，然后更新统计信息。示例 11-16 显示了这个命令的一个实例。

### 示例 11-16: OPTIMIZE TABLE 命令

```
mysql> OPTIMIZE TABLE film \G
***** 1. row *****
    Table: sakila.film
      Op: optimize
Msg_type: note
Msg_text: Table does not support optimize, doing recreate + analyze instead
***** 2. row *****
    Table: sakila.film
      Op: optimize
Msg_type: status
Msg_text: OK
2 rows in set (0.44 sec)
```

我们在这个结果集中看到两行信息。第一行表明无法执行 OPTIMIZE TABLE 命令，而是重

新创建表然后执行 `ANALYZE TABLE` 命令。第二行是 `ANALYZE TABLE` 命令运行的结果。

435 同 `ANALYZE TABLE` 命令一样, 执行过程中发生的任意异常事件都会在 `Msg_type` 字段体现, 值为 `info`、`Error` 或 `warning`, 这时 `Msg_text` 字段将会显示事件的额外信息。如果得到的结果不是 `status` 或 `OK`, 应该审查一下。



使用 InnoDB 的时候, 尤其当存在二级索引 (通常会导致碎片产生) 时, 可能看不到任何改进, 或者操作运行的时间很长, 除非使用 InnoDB 的 “快速索引创建” 选项, 但这取决于索引是如何创建的。可能并不适用于全部索引。

## 数据库优化的最佳实践

如前所述, 关于优化有许多很好的例子、技术和方法被世界上最好的数据库性能专家强烈推荐。因为监控是用来检测和诊断性能问题的, 我们将总结一下监控 MySQL 的最佳实践。

为了简化问题的同时避免那些具有争议的技术, 我们仅讨论提高数据库性能的一些普遍认同的最佳实践。建议读者参看前面提到的参考书以了解关于这些方法的细节。

### 谨慎而有效地使用索引

大多数数据库专家都知道索引的重要性以及它们是如何提高性能的。通常, 使用 `EXPLAIN` 命令是确定需要哪些索引的最好办法。虽然索引不足引起的问题是可以理解的, 但是有时候拥有太多的索引也会引起性能问题。

前面介绍 `EXPLAIN` 命令时我们看到, 创建太多索引, 或者索引很少用到或没有用, 都是有可能发生的。每个索引都会为表的每次删除和插入操作增加开销。有些时候, 索引过多且分布很广 (有很多值) 时, 会大大降低插入和删除的性能。它还会降低复制和恢复操作的性能。

应该定期检查索引, 确保它们都是有意义的并且被使用了。这些索引应该删除: 没有被使用、使用有限或分布很广的索引。通常可以使用规范化解决一些分布广的问题。

### 使用规范化, 但不要过度使用

许多学习计算机科学或相关学科的数据库专家可能对 C.J. Date 等描述的范式有美好的 (或噩梦式) 回忆。我们不再在这里重温这些知识, 而是讨论这些经验教训的影响。

规范化 (至少是第三范式) 是一个很好理解的标准方法。然而, 有时候你可能希望违反这些规则。

查找表的使用通常是规范化的产物，也就是说，你创建了一个特殊的表，这个表包含了频繁在其他表中用到的相关信息列表。然而，当使用那些经常被访问且分布有限（仅有几行，或值较小的有限行）的查找表时，会使系统性能降低。在这种情况下，每次用户查询信息的时候，必须使用 JOIN 操作获取完整数据。JOIN 操作的开销很大，而且频繁访问的数据会随着时间逐渐增加。为了减轻这种潜在的性能问题，可以使用枚举字段代替查找表存储数据。例如，使用枚举字段存储头发颜色，而不用创建表（尽管有些亚文化可能坚持要求这样，但是头发颜色类型真的很有限），这样可以避免使用 JOIN 操作。

例如，如果创建了一个子表包含头发颜色的所有可能值，主表中有一个字段的值是这个头发颜色表的索引。当执行查询从主表获取结果的时候，必须做 JOIN 操作获取头发颜色字段的值。如果用的是枚举字段，则不必使用 JOIN 操作从而性能得到提升。

另一个潜在问题是关于已计算字段的。通常情况下，我们不存储由其他数据形成的数据（如销售税或几个列的总和）。相反，这些已计算数据要么在数据检索过程中通过视图执行，要么在应用程序中执行。如果计算很简单，或者很少执行计算，这可能不是什么问题，但是如果计算很复杂而且需要频繁执行呢？如果是这样的话，执行这些计算就会浪费很多时间。缓解这个问题的办法是使用触发器计算值，然后将结果存储在表中。虽然从技术上讲会产生重复数据（规范化理论的一个大忌），但是它可以在需要执行大量计算的时候提高性能。

## 使用正确的存储引擎完成任务

MySQL 的最强大功能之一是它支持不同的存储引擎。存储引擎管理如何存储和恢复数据。MySQL 支持多个存储引擎，每个存储引擎都具有独特的功能和用途。数据库设计者通过使用最适合他们应用需求的存储引擎来改善数据库系统的性能。例如，如果你的环境需要事务控制应对高度活跃的数据库，请选择一个适合这个任务的存储引擎。再比如，你可能会发现有个视图或表常常被查询但是几乎没有任何更新（例如查找表），这时你可能希望使用某个存储引擎将这些数据存储在内存中，以便快速访问。

MySQL 的最新变化是将有些存储引擎改成了插件，还有些 MySQL 发行版默认仅启用某些存储引擎。执行 SHOW ENGINES 命令可查看启用了哪些存储引擎。示例 11-17 显示了一个 MySQL 典型安装中的存储引擎。

◀ 437

### 示例 11-17：存储引擎

```
mysql> SHOW ENGINES \G
```

```
***** 1. row *****
```

```
Engine: InnoDB
```

```
Support: YES
```

```
Comment: Supports transactions, row-level locking, and foreign keys
```

```

Transactions: YES
          XA: YES
    Savepoints: YES
***** 2. row *****
    Engine: MyISAM
    Support: DEFAULT
    Comment: Default engine as of MySQL 3.23 with great performance
Transactions: NO
          XA: NO
    Savepoints: NO
***** 3. row *****
    Engine: BLACKHOLE
    Support: YES
    Comment: /dev/null storage engine (anything you write to it disappears)
Transactions: NO
          XA: NO
    Savepoints: NO
***** 4. row *****
    Engine: CSV
    Support: YES
    Comment: CSV storage engine
Transactions: NO
          XA: NO
    Savepoints: NO
***** 5. row *****
    Engine: MEMORY
    Support: YES
    Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
          XA: NO
    Savepoints: NO
***** 6. row *****
    Engine: FEDERATED
    Support: NO
    Comment: Federated MySQL storage engine
Transactions: NULL
          XA: NULL
    Savepoints: NULL
***** 7. row *****
    Engine: ARCHIVE
    Support: YES
    Comment: Archive storage engine
Transactions: NO
          XA: NO
    Savepoints: NO

```



```
***** 8. row *****
Engine: MRG_MYISAM
Support: YES
Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
Savepoints: NO
8 rows in set (0.00 sec)
```

这个结果集包含了所有已知的存储引擎，它们是否被安装和配置（Support=YES），存储引擎功能的说明，以及它们是否支持事务、分布式事务（XA）或保存点（savepoint）等。

保存点是可以像事务一样使用的命名事件。建立一个保存点，然后释放（删除保存点）或从保存点开始回滚变更。参看在线 MySQL 参考手册可获取更多关于保存点的内容。

有这么多存储引擎供选择，在设计数据库的时候，对为了提高性能如何选择存储引擎就会很困惑。在 CREATE 语句中使用 ENGINE 参数为表选择存储引擎，使用 ALTER TABLE 命令更改存储引擎。

```
CREATE TABLE t1 (a int) ENGINE=InnoDB;
ALTER TABLE t1 ENGINE=MEMORY;
```

下面简单介绍一下每个存储引擎及其最适合的使用场合。

## InnoDB

作为第一个事务型存储引擎，InnoDB 也是默认的存储引擎。<sup>注1</sup>如果在 CREATE 语句中不指定 ENGINE 选项，则使用 InnoDB 引擎。需要事务支持的时候，总是应该选择这个存储引擎。InnoDB 和 NDB 是 MySQL 中目前仅有的事务型引擎。不同的产品线上有很多第三方存储引擎也能支持事务，但是只有 InnoDB 是“拆包即用”的。InnoDB 适用于高性能和事务处理环境。

## MyISAM

MyISAM 通常用在数据仓库、电子商务和企业应用中，这些环境下大部分操作都是读操作（称为只读为主，read-mostly）。MyISAM 使用高级缓存和索引机制提高数据检索和索引速度。当多种应用程序需要快速检索数据而不需要事务时，MyISAM 是个很好的选择。

439

## Blackhole

这个存储引擎非常有趣，它并不存储任何东西。实际上，正如它的名字所言——进

注1 InnoDB 在版本 5.5 中成为默认存储引擎。

去的数据永远不会返回。不开玩笑了。Blackhole 存储引擎满足一个特殊的需求。如果启用了二进制日志，SQL 语句将被写入日志，将 Blackhole 存储引擎作为复制拓扑中的中继（或代理）。在这种情况下，中继代理处理来自于 master 的数据，并将这些数据发送到它的 slave 上去，但是它本身并不存储任何数据。如果希望测试应用程序确实在写数据而又不希望在磁盘上存储任何数据，Blackhole 存储引擎很有用。

## CSV

CSV 存储引擎以表格形式创建、读取和写入逗号分隔值（CSV）文件。CSV 存储引擎的最佳用处是将结构化业务数据快速导入电子表格，但它不提供任何索引机制，在存储和转换日期/时间值（在查询中它们不遵循本地性原则）时也存在某些问题。如果想让其他应用程序以共同的格式共享或交换数据，CSV 存储引擎非常有用。鉴于它存储数据的效率不高，应该谨慎使用。



CSV 存储引擎用于写日志文件。例如，备份日志是 CSV 文件，其他应用程序可以使用 CSV 协议打开它（但是在服务器运行的时候不能打开）。

## Memory

这个存储引擎（有时被称为 HEAP）是内存中的存储器，使用哈希机制检索频繁使用的数据，从而检索更快。它访问数据的方式与其他存储引擎类似，但是由于数据存储在内存中，只在 MySQL 会话中有效，关机时数据被刷新并删除。Memory 存储引擎通常适用于频繁访问而很少更改静态数据（如查找表）的情况。例如，邮编列表、州和国家名字、分类列表等频繁访问而几乎不被更新的数据。此外，Memory 存储引擎还适用于那些利用快照技术访问分布数据或历史数据的数据库。

440

## Federated

创建参照多个数据库系统的单个表。Federated 存储引擎允许将多个数据库服务器的表连接起来。这个机制的目的类似于连接其他数据库系统中的数据表。Federated 存储引擎最适合分布式或数据集环境。它最有趣的特性是：不移动数据，也不要求远程表使用相同的存储引擎。



目前 Federated 存储引擎在 MySQL 的大部分发行版中是禁用的。参阅在线 MySQL 参考手册可获取更多详情。

## Archive

这个存储引擎以压缩格式存储大量数据。它最适合存储和检索大量很少访问的存档或历史数据。但是它不支持索引，且只能通过表扫描访问。因此，常规数据库存储和检索时不应该使用 Archive 存储引擎。

## Merge

这个存储引擎 (MRG\_MYISAM) 使用相同的结构 (表或模式) 将一组 MyISAM 表封装成单个表。因此，这些表按单个表的位置分区，但是没有使用额外的划分机制。所有的表都必须存放在同一个服务器上 (但不需要在同一个数据库中)。



当在合并表上执行 DROP 命令时，仅删除 Merge 规则，原始表并没有被改变。

Merge 存储引擎最棒的特点就是速度。将一个大表分割成许多不同的小表，存储在不同的磁盘上，把这些小表合并，然后同时访问它们。搜索和排序执行得更快，因为每个小表需要管理的数据变少了。另外，表的修复也更高效，因为修复多个更小且独立的表比修复一张大表更快、更容易。不幸的是，这种配置有几个缺点：

- 必须使用相同的 MyISAM 表组成一个合成表。
- 替换操作不可用。
- 索引比单表的索引效率低。

441

Merge 存储引擎最适合非常大的数据库 (VLDB) 应用，如数据仓库，其中数据存储在一个或多个数据库的多个表中。此外，Merge 存储引擎还可以用于解决数据划分问题：你想水平划分数据，但是不希望增加划分表选项的复杂性。

显然，由于可供选择的存储引擎很多，有可能所选择的存储引擎会影响性能，甚至有时会禁用某些解决方案。例如，如果创建表的时候从不指定存储引擎，MySQL 使用默认的存储引擎。如果没有手动设置，默认存储引擎将恢复为平台特定的默认存储引擎，有些平台上使用的是 MyISAM。这可能意味着无法优化查询表，或者由于不支持事务限制了应用程序的功能。所以，在设计或优化数据库时，最好多花点时间分析一下如何选择存储引擎。

## 通过查询缓存使用视图来更快获取结果

封装复杂查询，简化数据处理工作的一个很方便的方法是视图。使用视图可以水平地 (更少的列) 或垂直地 (在 SELECT 语句中使用 WHERE 子句) 限制数据。两者都很方便。当然，



复杂的视图同时使用这两种方法限定返回的结果集给用户，或者隐藏某些基表，或者执行了有效的 JOIN 操作。

使用视图来限定返回列在很多方面都很有用，有些你甚至没有想到。它不仅降低了数据的处理量，还可以避免不假思索地执行开销较大的 `SELECT *` 操作。如果这种类型的操作很多，那么应用程序将会处理太多数据，这不但影响应用程序的性能，而且影响服务器的性能，更重要的是，它会降低可用网络带宽。最好使用视图限定数据，隐藏对基表的访问，避免用户直接访问基表。

限定返回行数的视图还可以减少网络带宽，提高应用程序的性能。这些视图还可以避免滥用 `SELECT *` 查询。以这种方式使用视图还需要多一点规划，因为你的目的是创建有意义的数据子集。需要了解数据库的需求并理解查询，使用正确的 `WHERE` 子句。

442

付出一点努力，你就可以创建既有水平限定又有垂直限定的视图，从而确保应用程序仅处理那些需要的数据。数据迁移越少，在相同的时间内应用程序可以处理的数据就越多。

也许视图的最佳用处是消除低性能的 JOIN 操作，尤其是规范化模式很复杂的时候。对于用户而言，可能并不清楚如何合并表以形成有意义的结果集。实际上，为了获取更好的性能，DBA 的大部分工作都与低性能的 JOIN 操作相关。有时候挺微不足道的——比如，使 JOIN 操作处理更少的行——但是很多时候，改善响应时间很重要。

在 MySQL 中使用查询缓存的时候，视图也很有用。查询缓存存储频繁使用（访问）的查询结果。通过使用提供标准结果集的视图可以提高结果被缓存的概率，从而使检索更高效。

通过一些设计工作和明智地使用视图，可以提高性能。需要花时间去检查迁移的数据量（列的数量和行的数量）和应用程序中使用 JOIN 的查询，然后创建限定数据的视图，识别最有效的 JOIN 操作，并把它们封装在视图中。想象一下，如果用户都在执行高效的 JOIN 操作，你该多轻松。

## 使用约束

约束是另一个改善性能问题的工具。这里不讨论使用约束有哪些限制，我们认为约束是一个标准的做法，而不是事后补救的策略。

MySQL 有多种类型的约束，包括：

- 唯一索引
- 主键
- 外键



- 枚举值
- 集合
- 默认值
- NOT NULL 选项

我们已经讨论过如何使用索引和过度使用索引。索引有助于提高数据检索速度，使得系统更快地存储和查找数据。



外键是一种约束形式，与性能并不直接相关。但是，外键能够保护参照完整性。应该注意更新具有大量外键的表，或者执行级联操作，会对性能产生影响。目前，只有 InnoDB 支持外键。关于更多外键的信息，参见在线 MySQL 参考手册。

443

MySQL 中的集合（sets）与枚举值类似，在某个字段中限定值。可以使用集合存储数据属性信息，而不需要使用主表 / 细节表。这样不仅可以节省表空间（集合值是基于位的），而且不需要访问其他表。

使用 DEFAULT 选项为字段提供默认值，能够有效避免数据结构差的问题。例如，如果表中有个数值型字段，其值要用来计算，确保当这个字段未知时，用默认值代替它。大部分数据类型都可以设置默认值。还可以在日期和时间字段上使用默认时间，这样可以避免产生无效日期时间值。更重要的是，默认值使得应用程序不需要提供值（或者使用不太可靠的方法要求用户提供值），从而减少了发送到服务器的数据量。

要指定字段必须有值，还应该考虑 NOT NULL 选项。在执行 INSERT 操作时，如果没有给 NOT NULL 字段提供任何值，INSERT 语句将执行失败。这保证了数据的完整性，确保所有重要字段都有值。而且空值也会使得这些字段上的查询变慢。

## 使用 EXPLAIN、ANALYZE 和 OPTIMIZE

前面已经讨论了这些命令的好处，这里将这些命令的最佳实践罗列出来，以提醒你这些工具对诊断和调优很重要。在不发生错误的前提下可经常使用它们，但是请小心使用。具体来说，合理使用 ANALYZE 和 OPTIMIZE，而不是有规律地定期使用。我们遇到有些管理员晚上运行这些命令，有时候可以保证这一点，但通常这是无法保证的，会产生不必要的表副本（就像前面的实例那样）。因此，强制系统定期复制数据就是浪费时间，而且会导致操作过程中访问受限。

现在已经讨论了如何监控和提高 MySQL 查询性能，让我们来看一下关于性能审查的最佳实践。

## 提高性能的最佳实践

诊断和改善数据库性能的详细内容在专门讨论这个主题的书上有介绍，事实上，这些内容涉及很多页。

为了本书内容的完整性，仅作为一般的参考，这一节我们介绍处理性能异常的几个最佳实践方法，以作为指南性的检查清单。我们按照常见问题分类讨论这些实践。

### 一切都很慢

当整个系统性能很差时，必须关注系统是如何运行的，从操作系统开始。使用下面介绍的方法识别和改进系统的性能：

- 检查硬件问题。
- 改善硬件环境（例如，添加内存）。
- 考虑将数据迁移到独立的磁盘上。
- 检查操作系统的配置是否正确。
- 考虑将有些应用迁移到其他服务器上。
- 考虑横向扩展的复制。
- 优化服务器性能。

### 查询慢

使用下面的方法可改善慢查询日志中的任何查询，以及那些由用户或开发者识别出来的有问题的查询：

- 规范化数据库模式。
- 使用 EXPLAIN 识别丢失或不正确的索引。
- 使用 benchmark() 函数测试部分查询。
- 考虑重写查询。
- 对标准查询使用视图。
- 使用查询缓存进行测试（这点并不是对所有查询都有效，或者取决于访问模式的频率）。



不论复制查询是否会被写入 master 中的慢查询日志，复制 slave 都不会将它写入慢查询日志。

## 应用慢

如果应用程序出现性能问题，应该检查应用组件，以确认问题出在哪里。也许你会发现只有一个模块有问题，但是有时候可能更严重。下面介绍的方法可以帮助你识别和解决应用程序的性能问题：

- 开启查询缓存。
- 如果查询缓存已经启用了，关闭它有时候可能会改善某些查询。使用 DEMAND 模式和 SELECT SQL\_CACHE，按需使用查询缓存。
- 考虑并优化存储引擎。
- 确定是否是服务器或操作系统的问题。
- 定义应用程序的基准测试，并将它与已知基准进行比较。
- 检查内部（在应用程序中编写的）查询，并最大化它们的性能。
- 分而治之——一次只检查一个部分。
- 使用分区来分散数据。
- 检查各个分区的索引。

## 复制慢

前面提过，与复制相关的性能问题，通常与数据库和服务器性能问题无关。使用以下方法可诊断复制的性能问题：

- 确保网络峰值性能最佳。
- 确保服务器配置正确。
- 优化数据库。
- 限制 master 的更新。
- 将读操作分散到多个 slave 中。
- 检查 slave 的复制延迟。
- 定期维护日志（二进制日志和中继日志）。
- 如果带宽有限，使用压缩。
- 使用包容性和排他性日志选项，最小化复制内容。

446

## 小结

监控 MySQL 服务器有很多事情需要做。我们已经讨论了用于监控服务器的基本 SQL 命令、*mysqladmin* 命令行实用程序、基础测试套件和 MySQL 工作台。我们还介绍了提高数据库性能的一些最佳实践。

现在你已经了解了操作系统监控、数据库性能、MySQL 监控和基准测试的基本知识，而且已经拥有能够成功优化服务器性能的工具和知识。

Joel 一边笑，一边在写 Susan 那个嵌套查询问题的报告。他花了好几个小时检查日志文件才找出问题，向开发人员解释完查询的开销之后，他们同意更改查询，使用存储在内存表中的查找表。Joel 觉得老板应该会很满意他的构思。刚点击发送，这时老板出现在他的办公室门口。

“Joel！”

尽管知道是 Summerson 先生，Joel 还是被吓了一跳。“我已经解决了市场部的应用问题，先生。”他赶紧说。

“太好了，我很期待着你的方案。”

Joel 不确定老板是否理解他邮件中提到的技术部分，但是他知道，如果不向老板解释，老板也会一直问的。

Summerson 先生点了点头，然后走了。Joel 打开了来自于西雅图的 Phil 的邮件，这个邮件是抱怨复制问题的。Joel 马上意识到问题远比他现在了解的要严重。



## 监控存储引擎

Joel 正在喝拿铁，这时办公室中的电话铃响了。他吓了一跳，因为从早上到现在电话还没响过。他拿起电话筒，听到了发动机的声音。他以为电话打错了，迟疑地说：“喂？”

“Joel！联系上你太好了！”原来是 Summerson 先生从车里打来的。

“是的，先生。”

“我正在去机场的路上，去接见西雅图办事处的销售人员。我想让你看看新的应用数据库。西雅图的开发人员说我们需要搞个更好的配置来提高性能。”

Joel 就知道会发生这种事情。他知道一点 MyISAM 和 InnoDB，但是并不熟悉监控，更不了解性能调优。“我会去做的，先生。”

“太好了，谢谢你，Joel。我会给你发邮件的。”Joel 还没来得及回答电话就被挂断了。

Joel 喝完了最后一点拿铁，然后开始阅读关于 MySQL 存储引擎的知识。

现在你已经知道服务器何时运行良好（以及何时运行不好），那么怎么知道存储引擎的运行情况如何呢？如果有一个或多个事务型数据库，或者需要存储引擎快速高效地处理查询，那么就需要监控存储引擎。本章讨论高级存储引擎监控，重点关注提高存储引擎的性能，主要介绍两个最流行的存储引擎：InnoDB 和 MyISAM。我们将分别介绍这两个存储引擎，并就如何改善性能提出一些实用的建议。

### InnoDB

InnoDB 存储引擎是 MySQL（版本 5.5）的默认存储引擎。InnoDB 提供了高可用和高性能的事务型操作，完全支持 ACID 事务。InnoDB 被证明是非常可靠的，而且一直在改善中。

最新的改进包括支持多核处理器，改善了内存分配，以及更细粒度的性能调优能力。在线参考手册给出了关于 InnoDB 存储引擎全部功能的详细解释。

InnoDB 存储引擎有很多优化选项，深入讲解所有选项和相关技术足以写一本书。例如，控制 InnoDB 行为的变量有 50 个，表示性能和状态相关的元数据的状态变量有 40 多个。本节我们讨论如何监控 InnoDB 存储引擎，并关注提高性能的关键问题。

这里不深入讨论这些领域，而是仅仅对于以下性能提升方法提供策略：

- 使用 SHOW ENGINE 命令
- 使用 InnoDB 监视器
- 监控日志文件
- 监控缓冲池
- 监控表空间
- 使用 INFORMATION\_SCHEMA 表
- 使用 PERFORMANCE\_SCHEMA 表
- 考虑其他参数
- 对 InnoDB 进行故障排除

下面我们简单讨论这些问题。但是，在开始讨论前，让我们先简单了解一下 InnoDB 的架构特性。

InnoDB 存储引擎的架构非常复杂，专门为高并发性和大量事务性活动而设计。它有许多高级功能，应该在尝试改进性能前优先考虑这些功能。我们主要关注那些可以被监控和改进的功能，包括索引、缓冲池、日志文件和表空间。

InnoDB 表使用的索引是聚集索引。即使未指定索引，InnoDB 也会为每行分配一个内部值，从而使用聚集索引。聚集索引是一种数据结构，它不仅存储索引，还存储数据本身。也就是说，一旦定位到索引中的某个值，就可以直接检索数据而无须额外的磁盘寻道。当然，主键索引或者表的第一个唯一索引都采用聚集索引创建。

如果创建了二级索引，聚集索引的关键字（主键、唯一键或者行 ID）也会存在二级索引中。这样可以快速按关键字查找和快速获取聚集索引中的原始数据。这也意味着如果使用主键列扫描二级索引，则查询只需要用二级索引就可以获取数据。

缓冲池是用于管理事务和读写磁盘数据的缓存机制，如果配置得当，可以减少磁盘访问。缓冲池同时还是崩溃恢复的一个重要组成部分，因为缓冲池内的信息将会定期被写入磁盘（例如关机时）。默认情况下，缓冲池状态保存在 `ib_buffer_pool` 文件中，位于 InnoDB 数据文件所在的目录下。因为状态是一个内存组件，必须监控缓冲池的有效性，

保证配置的正确性。

InnoDB 也使用缓冲池来存储数据变更和事务。InnoDB 缓存变更的方式是，将数据变更保存到缓冲池中的数据页（块）中。每次引用数据页的时候，该页都会放到缓冲池中，如果该页发生改变，则标记为“脏页”。然后，这个变更被写入到磁盘以更新数据，并向重做日志写入一个副本。这些日志文件的名字为 `ib_logfile0` 或 `ib_logfile1`。在 MySQL 服务器的数据目录中可以看到这些文件。



更多有关刷新缓冲池的配置和控制，参见在线 MySQL 参考手册中“改善缓冲池的刷新”一节。

InnoDB 存储引擎使用两种基于磁盘的机制存储数据，即日志文件和表空间。在关机或死机之前，InnoDB 还会使用这些日志来重建（或重做）数据修复。在程序启动时，InnoDB 读取日志并自动将脏页写入磁盘，从而在系统崩溃前恢复缓冲中的更新。

## 单独的表空间

有一个最新的性能功能是允许重做日志以单独表空间的形式保存。由于重做日志对于长时间运行的事务来说会消耗大量空间，将重做日志放在单独或多个表空间，能够减少系统表空间的大小。要把重做日志放在单独的表空间，将 `--innodb_undo_tablespaces` 配置选项设置为大于零的值。还可以通过 `--innodb_undo_directory` 选项指定重做日志的表空间。MySQL 以 `innodb $n$`  的形式为重做日志表空间命名，其中  $n$  是以若干 0 开头的连续整数。

450

表空间是 InnoDB 用来组织与机器无关的文件的工具，包括数据、索引及回滚机制（用于回滚事务）。默认情况下，所有表共享一个表空间（称为共享表空间）。共享表空间不会自动扩展成多个文件。默认情况下，一个表空间只占据单个文件，该文件随数据增加而增长。指定 `autoextend` 选项可以允许表空间创建新的文件。

还可以将表存储在它们自己的表空间中（称为独立表空间）。独立表空间包含数据和表的索引。虽然仍然有一个中心的 InnoDB 文件，但独立表空间能够将数据隔离在不同的文件（表空间）中。这些表空间可以自动扩展成多个文件，使得表中可以存储更多的数据，超出了操作系统可以处理的数据量。你还可以将表空间划分为多个文件，然后存储在不同的磁盘上。



使用 `innodb_file_per_table` 可为每个表创建单独的表空间。在设置该选项之前创建的表仍然存储在共享表空间中。使用此命令只影响新创建的表，而不会减少共享表空间的内容，或节省那些已经分配到共享表空间的空间。如果要对已有的表生效，在开启 `innodb_file_per_table` 之后使用 `ALTER TABLE... ENGINE=INNODB` 命令。

## 使用 SHOW ENGINE 命令

`SHOW ENGINE INNODB STATUS` 命令（又称 InnoDB 监视器）显示有关 InnoDB 存储引擎状态的统计和配置信息。这是查看 InnoDB 信息的标准方法。该命令显示的统计列表很长很全面。示例 12-1 节选了标准 MySQL 安装上该命令后的运行情况。

示例12-1: `SHOW ENGINE INNODB STATUS` 命令

```
mysql> SHOW ENGINE INNODB STATUS \G
***** 1. row *****
Type: InnoDB
Name:
Status:
=====
2013-01-08 20:50:16 11abaa000 INNODB MONITOR OUTPUT
=====
Per second averages calculated from the last 3 seconds
-----
BACKGROUND THREAD
-----
srv_master_thread loops: 1 srv_active, 0 srv_shutdown, 733 srv_idle
srv_master_thread log flush and writes: 734
-----
SEMAPHORES
-----
OS WAIT ARRAY INFO: reservation count 2
OS WAIT ARRAY INFO: signal count 2
Mutex spin waits 1, rounds 19, OS waits 0
RW-shared spins 2, rounds 60, OS waits 2
RW-excl spins 0, rounds 0, OS waits 0
Spin rounds per wait: 19.00 mutex, 30.00 RW-shared, 0.00 RW-excl
-----
TRANSACTIONS
-----
Trx id counter 1285
Purge done for trx's n:o < 0 undo n:o < 0 state: running but idle
History list length 0
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0, not started
MySQL thread id 3, OS thread handle 0x11abaa000, query id 32 localhost
```



127.0.0.1 root init  
SHOW ENGINE INNODB STATUS

-----  
FILE I/O

-----  
I/O thread 0 state: waiting for i/o request (insert buffer thread)  
I/O thread 1 state: waiting for i/o request (log thread)  
I/O thread 2 state: waiting for i/o request (read thread)  
I/O thread 3 state: waiting for i/o request (read thread)  
I/O thread 4 state: waiting for i/o request (read thread)  
I/O thread 5 state: waiting for i/o request (read thread)  
I/O thread 6 state: waiting for i/o request (write thread)  
I/O thread 7 state: waiting for i/o request (write thread)  
I/O thread 8 state: waiting for i/o request (write thread)  
I/O thread 9 state: waiting for i/o request (write thread)  
Pending normal aio reads: 0 [0, 0, 0, 0] , aio writes: 0 [0, 0, 0, 0] ,  
ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0  
Pending flushes (fsync) log: 0; buffer pool: 0  
171 OS file reads, 5 OS file writes, 5 OS fsyncs  
0.00 reads/s, 0 avg bytes/read, 0.00 writes/s, 0.00 fsyncs/s

-----  
INSERT BUFFER AND ADAPTIVE HASH INDEX

-----  
Ibuf: size 1, free list len 0, seg size 2, 0 merges  
merged operations:  
insert 0, delete mark 0, delete 0  
discarded operations:  
insert 0, delete mark 0, delete 0  
Hash table size 276671, node heap has 0 buffer(s)  
0.00 hash searches/s, 0.00 non-hash searches/s

---  
LOG

---  
Log sequence number 1625987  
Log flushed up to 1625987  
Pages flushed up to 1625987  
Last checkpoint at 1625987  
0 pending log writes, 0 pending chkp writes  
8 log i/o's done, 0.00 log i/o's/second

-----  
BUFFER POOL AND MEMORY

-----  
Total memory allocated 137363456; in additional pool allocated 0  
Dictionary memory allocated 55491  
Buffer pool size 8191

```

Free buffers 8034
Database pages 157
Old database pages 0
Modified db pages 0
Pending reads 0
Pending writes: LRU 0, flush list 0 single page 0
Pages made young 0, not young 0
0.00 youngs/s, 0.00 non-youngs/s
Pages read 157, created 0, written 1
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
No buffer pool page gets since the last printout
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 157, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
-----
ROW OPERATIONS
-----
0 queries inside InnoDB, 0 queries in queue
0 read views open inside InnoDB
Main thread id 4718366720, state: sleeping
Number of rows inserted 0, updated 0, deleted 0, read 0
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
-----
END OF INNODB MONITOR OUTPUT
=====

1 row in set (0.00 sec)

```

SHOW ENGINE INNODB MUTEX 命令展示了 InnoDB 的互斥体 (MUTEX) 信息, 对存储引擎中的线程调优很有帮助。示例 12-2 节选了标准 MySQL 安装上该命令后的运行情况。

#### 453 示例12-2: SHOW ENGINE INNODB MUTEX 命令

```
mysql> SHOW ENGINE INNODB MUTEX;
```

```

+-----+-----+-----+
| Type | Name | Status |
+-----+-----+-----+
| InnoDB | trx/trx0rseg.c:167 | os_waits=1 |
| InnoDB | trx/trx0sys.c:181 | os_waits=7 |
| InnoDB | log/log0log.c:777 | os_waits=1003 |
| InnoDB | buf/buf0buf.c:936 | os_waits=8 |
| InnoDB | fil/fil0fil.c:1487 | os_waits=2 |
| InnoDB | srv/srv0srv.c:953 | os_waits=101 |
| InnoDB | log/log0log.c:833 | os_waits=323 |
+-----+-----+-----+
7 rows in set (0.00 sec)

```

Name 列显示了创建互斥体的源文件和行号。Status 列显示了互斥体在操作系统上等待的次数（例如，os\_waits=5）。如果源代码是使用 UNIV\_DEBUG 指令编译的，该列的值可能是下面的其中一个：

count

请求互斥体的次数。

spin\_waits

自旋锁运行的次数。

os\_waits

互斥体在操作系统上等待的次数。

os\_yields

线程放弃时间片并返回操作系统的次数。

os\_wait\_times

互斥体等待操作系统的总时间。

SHOW ENGINE INNODB STATUS 命令显示了直接来自 InnoDB 存储引擎的很多信息。虽然这些信息没有格式化（即不是显示整齐的行和列），但是有许多工具可以将它们重新排列显示。例如，InnoTop（见第 11 章的“InnoTop”一节）命令就是使用这种方式获取数据的。

## 使用 InnoDB 监视器

InnoDB 存储引擎是唯一支持直接监控的本地存储引擎。InnoDB 背后有一个称为监视器（monitor）的特殊机制，它为服务器和客户端工具收集和报告统计信息。下面各项（以及大多数第三方工具）与 InnoDB 监控工具交互，因此 InnoDB 监视器通过 MySQL 服务器监控以下内容：

- 表和记录锁
- 锁等待
- 信号量等待
- 文件 I/O 请求
- 缓冲池
- 清除和插入缓冲合并活动

454

通过 SHOW ENGINE INNODB STATUS 命令自动连接 InnoDB 监视器，并显示监视器产生的信息。不过，还可以在 MySQL 中通过创建特殊的表，直接从 InnoDB 监视器得到这些信息。

这些表的真实结构及存储在什么地方并不重要（如果使用 `ENGINE=INNODB` 语句的话）。一旦创建，每个表中的数据都会转储到标准错误。通过 MySQL 错误日志可以查看这些信息。例如，在 Mac OS X 上默认安装 MySQL 时有个名为 `/usr/local/mysql/data/localhost.err` 的错误日志。在 Windows 上通过使用 `--console` 选项启动 MySQL，还可以在控制台显示这些输出。要启动 InnoDB 监视器，请在数据库中创建这些表：

```
mysql> SHOW TABLES LIKE 'innodb%';
```

```
+-----+
| Tables_in_test (innodb%) |
+-----+
| innodb_lock_monitor      |
| innodb_monitor           |
| innodb_table_monitor     |
| innodb_tablespace_monitor |
+-----+
4 rows in set (0.00 sec)
```

要关闭监视器，只需删除表即可。监视器每隔 15 秒自动重新生成数据。



重启会删除表。为了重启后继续监控，必须重新创建这些表。

每个监视器提供以下数据：

#### innodb\_monitor

标准监视器显示的信息与 SQL 命令相同。示例 12-1 给出了监视器输出结果的实例。

455

SQL 命令与 `innodb_monitor` 输出之间唯一的区别是：`innodb_monitor` 将结果以格式化的形式输出到标准错误，就像 MySQL 客户端垂直显示信息那样。

#### innodb\_lock\_monitor

锁监视器显示的信息与 SQL 命令相同，不过还包括锁的信息。这个报告可以用来检测死锁，发现并发问题，参见示例 12-3。

#### 示例12-3: InnoDB锁监视器报告

```
-----
TRANSACTIONS
-----
Trx id counter 2E07
Purge done for trx's n:o < 2C02 undo n:o < 0
History list length 36
```



LIST OF TRANSACTIONS FOR EACH SESSION:

---TRANSACTION 2E06, not started

mysql tables in use 1, locked 1

MySQL thread id 3, OS thread handle 0x10b2f3000, query id 30 localhost root

show engine innodb status

#### innodb\_table\_monitor

表监视器生成内部数据字典的详细报告。示例 12-4 显示了该报告的一个节选（为便于阅读，这里进行了格式化）。请注意每个表的扩展数据，包括字段定义、索引、行号、外键及其他更多的信息。在诊断表问题或者想了解索引细节时，可使用该报告。

#### 示例12-4: InnoDB表监视器报告

```
=====
2013-01-08 21:11:00 11dc5f000 INNODB TABLE MONITOR OUTPUT
=====
-----
TABLE: name SYS_DATAFILES, id 14, flags 0, columns 5, indexes 1, appr.rows 9
COLUMNS: SPACE: DATA_INT len 4; PATH: DATA_VARCHAR prtype 524292 len 0;
DB_ROW_ID: DATA_SYS prtype 256 len 6; DB_TRX_ID: DATA_SYS prtype 257 len 6;
DB_ROLL_PTR: DATA_SYS prtype 258 len 7;
INDEX: name SYS_DATAFILES_SPACE, id 16, fields 1/4, uniq 1, type 3
root page 308, appr.key vals 9, leaf pages 1, size pages 1
FIELDS: SPACE DB_TRX_ID DB_ROLL_PTR PATH
-----
...
-----
END OF INNODB TABLE MONITOR OUTPUT
=====
```

#### innodb\_tablespace\_monitor

◀ 456

显示共享表空间的扩展信息，包括文件段的列表，它还验证表空间分配的数据结构。该报告可能相当详细而且很长，因为它列出了表空间的所有细节信息。示例 12-5 显示了该报告的节选。

#### 示例12-5: InnoDB表空间监视器报告

```
=====
2013-01-08 21:11:00 11dc5f000 INNODB TABLESPACE MONITOR OUTPUT
=====
FILE SPACE INFO: id 0
size 768, free limit 576, free extents 3
not full frag extents 1: used pages 13, full frag extents 3
first seg id not used 180
SEGMENT id 1 space 0; page 2; res 2 used 2; full ext 0
```

```

fragm pages 2; free extents 0; not full extents 0: pages 0
SEGMENT id 2 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 3 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 4 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 5 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 6 space 0; page 2; res 0 used 0; full ext 0
fragm pages 0; free extents 0; not full extents 0: pages 0
SEGMENT id 7 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 8 space 0; page 2; res 0 used 0; full ext 0
fragm pages 0; free extents 0; not full extents 0: pages 0
SEGMENT id 9 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 10 space 0; page 2; res 0 used 0; full ext 0
fragm pages 0; free extents 0; not full extents 0: pages 0
SEGMENT id 11 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 12 space 0; page 2; res 0 used 0; full ext 0
fragm pages 0; free extents 0; not full extents 0: pages 0
SEGMENT id 13 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 14 space 0; page 2; res 0 used 0; full ext 0
fragm pages 0; free extents 0; not full extents 0: pages 0
SEGMENT id 15 space 0; page 2; res 160 used 160; full ext 2
fragm pages 32; free extents 0; not full extents 0: pages 0
SEGMENT id 16 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 17 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 18 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 19 space 0; page 2; res 1 used 1; full ext 0
457 fragm pages 1; free extents 0; not full extents 0: pages 0
SEGMENT id 20 space 0; page 2; res 1 used 1; full ext 0
fragm pages 1; free extents 0; not full extents 0: pages 0

```

...

NUMBER of file segments: 179

Validating tablespace

Validation ok

-----  
END OF INNODB TABLESPACE MONITOR OUTPUT  
=====

可见，InnoDB 监视器报告了相当多的详细信息。长时间开启 InnoDB 监视器会向日志文件添加大量数据信息。

## 监控日志文件

InnoDB 日志文件在你的数据和操作系统之间缓冲数据，所以保证日志文件正常运行可以获得良好的性能。查看以下系统状态变量，可以直接监控这些日志文件：

```
mysql> SHOW STATUS LIKE 'InnoDB%log%';
```

Variable_name	Value
InnoDB_log_waits	0
InnoDB_log_write_requests	0
InnoDB_log_writes	2
InnoDB_os_log_fsyncs	5
InnoDB_os_log_pending_fsyncs	0
InnoDB_os_log_pending_writes	0
InnoDB_os_log_written	1024

有些信息我们已经在 InnoDB 监视器的显示结果中见过了，不过使用以下状态变量还能获取日志文件的详细信息：

### Innodb\_log\_waits

当日志文件太小（即没有足够空间存储所有数据）时，操作必须等待日志刷新的等待时间计数器。如果该值开始增加并长时间大于零（除批量操作以外），可能需要增加日志文件的大小。

### Innodb\_log\_write\_requests

写日志请求的数量。

### Innodb\_log\_writes

数据被写入日志的次数。

### Innodb\_os\_log\_fsyncs

操作系统文件同步的次数（即 `fsync()` 方法调用）。

Innodb\_os\_log\_pending\_fsyncs

挂起的文件同步请求的数量。如果该值开始增加并长期大于零，可能需要检查磁盘访问问题。

Innodb\_os\_log\_pending\_writes

挂起的日志写请求的次数。如果该值开始增加并长期大于零，可能需要检查磁盘访问问题。

Innodb\_os\_log\_written

写入日志的字节总数。

所有这些选项都是数值信息，可以在 MySQL 工作台中创建自定义图表显示它们。

## 监控缓冲池

缓冲池是 InnoDB 缓存频繁访问数据的地方。缓冲池中数据的任何变更也会被缓存。缓冲池还存储当前事务的信息。因此，缓冲池是关乎性能的关键机制。

使用 `SHOW ENGINE INNODB STATUS` 命令可查看缓冲池的行为信息，如示例 12-1 所示。为了方便查看，这里重复列一下例子中的缓冲池和内存部分：

```
-----  
BUFFER POOL AND MEMORY  
-----
```

```
Total memory allocated 138805248; in additional pool allocated 0  
Dictionary memory allocated 70560  
Buffer pool size 8192  
Free buffers 760  
Database pages 6988  
Modified db pages 113  
Pending reads 0  
Pending writes: LRU 0, flush list 0, single page 0  
Pages read 21, created 6968, written 10043  
0.00 reads/s, 89.91 creates/s, 125.87 writes/s  
Buffer pool hit rate 1000 / 1000  
LRU len: 6988, unzip_LRU len: 0  
I/O sum[9786]:cur[259], unzip sum[0]:cur[0]
```

这个报告中需要重点注意以下几项（我们稍后再详细讨论状态变量）：

459

空闲缓冲区

空的、可用于缓冲数据的缓冲段个数。



已修改的页

已经发生变化的页（脏页）数。

待处理的读请求

等待中的读请求个数，该值应该保持在低水平。

待处理的写请求

等待中的写请求个数，该值应该维持在低水平。

命中率

缓冲区成功命中的请求个数与总请求数之间的比例，这个比值最好接近 1 : 1。

还有很多状态变量。下面显示了 InnoDB 缓冲池的状态变量：

```
mysql> SHOW STATUS LIKE 'InnoDB%buf%';
```

Variable_name	Value
InnoDB_buffer_pool_pages_data	21
InnoDB_buffer_pool_pages_dirty	0
InnoDB_buffer_pool_pages_flushed	1
InnoDB_buffer_pool_pages_free	8171
InnoDB_buffer_pool_pages_misc	0
InnoDB_buffer_pool_pages_total	8192
InnoDB_buffer_pool_read_ahead_rnd	0
InnoDB_buffer_pool_read_ahead_seq	0
InnoDB_buffer_pool_read_requests	558
InnoDB_buffer_pool_reads	22
InnoDB_buffer_pool_wait_free	0
InnoDB_buffer_pool_write_requests	1

缓冲池有很多状态变量用于显示关于缓冲池性能的主要统计信息，如缓冲池中的页状态、缓冲池的读写信息，以及缓冲池中读写等待的频率。下面详细介绍各个状态变量：

**InnoDB\_buffer\_pool\_pages\_data**

含有数据的页数，包括不变的页和更改过的页（即脏页）。

**InnoDB\_buffer\_pool\_pages\_dirty**

更改过的页（即脏页）的数目。

**InnoDB\_buffer\_pool\_pages\_flushed**

缓冲池页面被刷新的次数。

InnoDB\_buffer\_pool\_pages\_free

空（空闲）页的数目。

InnoDB\_buffer\_pool\_pages\_misc

InnoDB 引擎执行管理性工作用到的页数，其计算方式如下：

$$X = \text{InnoDB\_buffer\_pool\_pages\_total} - \text{InnoDB\_buffer\_pool\_pages\_free} - \text{InnoDB\_buffer\_pool\_pages\_data}$$

InnoDB\_buffer\_pool\_pages\_total

缓冲池中的总页数。

InnoDB\_buffer\_pool\_read\_ahead\_rnd

InnoDB 扫描大数据块时发生随机预读的数量。

InnoDB\_buffer\_pool\_read\_ahead\_seq

顺序全表扫描时发生的顺序预读的数量。

InnoDB\_buffer\_pool\_read\_requests

逻辑读请求的次数。

InnoDB\_buffer\_pool\_reads

直接从磁盘中逻辑读取（而不是从缓冲池读）的次数。

InnoDB\_buffer\_pool\_wait\_free

如果缓冲池忙或者没有空闲页，InnoDB 可能需要等待页面刷新。该值表示等待的次数。如果这个值增加且始终大于 0，可能是缓冲池过小或磁盘访问出现问题。

InnoDB\_buffer\_pool\_write\_requests

写入 InnoDB 缓冲池的次数。

所有这些选项都是数值信息，可以在 MySQL 工作台中创建自定义图表显示它们。

## 监控表空间

如果 InnoDB 可以在运行缓慢时扩展表空间，那么 InnoDB 的表空间基本可以自给自足。使用 `autoextend` 选项配置 `innodb_data_file_path` 变量，可以自动扩展 InnoDB 表空间。

461 例如，MySQL 安装时默认的共享表空间大小为 10MB，可以自动扩展为更多文件：

```
--innodb_data_file_path=ibdata1:10M:autoextend
```

详见在线 MySQL 参考手册的“InnoDB 配置”一节。

使用 `SHOW ENGINE STATUS INNODB` 命令可查看当前的表空间配置信息。打开 InnoDB 表空间监视器能够查看表空间的详细信息（详见在线 MySQL 参考手册中的“使用表空间监视器”一节）。

## 使用 INFORMATION\_SCHEMA 表

*INFORMATION\_SCHEMA* 数据库中含有大量 InnoDB 表。从技术上说，这些表是视图，因为它们的数据并不是真的存储在磁盘上的，而是在查询时生成的。这些表提供了另一种监控 InnoDB 的方法，并为管理员提供性能信息。5.5 及其之后的版本默认会有这些表。

这些表用于监控压缩、事务和锁等。这里我们简单介绍其中几个表：

### INNODB\_CMP

显示压缩表的详细信息和统计信息。

### INNODB\_CMP\_RESET

与 *INNODB\_CMP* 表显示相同的信息，但是查询表的时候会重置统计信息，从而可以定期（如每小时、每天等）跟踪统计信息。

### INNODB\_CMPMEM

显示在缓冲池中压缩使用情况的详细信息和统计信息。

### INNODB\_CMPMEM\_RESET

与 *INNODB\_CMPMEM* 表显示相同的信息，但是查询表的时候会重置统计信息，从而可以定期（如每小时、每天等）跟踪统计信息。

### INNODB\_TRX

显示所有事务的详细信息和统计信息，包括当前正在处理的状态和查询。

### INNODB\_LOCKS

显示事务请求的所有锁的详细信息和统计信息。描述每个锁的状态、模式、类型等信息。

### INNODB\_LOCK\_WAITS

显示事务请求的所有被阻塞的锁的详细信息和统计信息，描述每个锁的状态、模式、类型和阻塞事务。



每个表的完整描述，包括字段和使用示例，参见在线参考手册可获取更多信息。

使用压缩表可监控表的压缩信息，包括页大小、使用哪些页、压缩时间和解压时间等详细信息。如果使用了压缩并希望压缩带来的开销不会影响数据库服务器的性能，那么这些信息将是重要的监控对象。

可使用事务和锁定表来监控事务。这是一个非常有用的工具，可以保证事务型数据库顺利运行。更重要的是，它可以精确地确定各个事务所在的状态，以及哪些事务被阻塞，哪些处于锁定状态。这些信息对于诊断复杂的事务问题（如死锁或性能低下）也是很重要的。



旧版本的 InnoDB，特别是 MySQL 5.1 版本，是以插件的形式构建的存储引擎。如果使用的是旧版本的 InnoDB 存储引擎插件，还需要访问 `INFORMATION_SCHEMA` 数据库中的 7 个特殊的表。必须单独安装 `INFORMATION_SCHEMA` 表。更多详情可参考 InnoDB 插件文档。

## 使用 PERFORMANCE\_SCHEMA 表

从 MySQL 5.5 版本开始，InnoDB 支持 MySQL 服务器的 `PERFORMANCE_SCHEMA` 功能了。第 11 章介绍过 `PERFORMANCE_SCHEMA`。对 InnoDB 来说，这意味着能够监控 InnoDB 子系统的内部行为，从而能够利用源码对 InnoDB 进行调优。严格来说，并不一定要阅读 InnoDB 源码才能使用 `PERFORMANCE_SCHEMA` 表进行 InnoDB 调优，不过了解源码的专业用户能够获取更多精准性能调优的知识。不过这是需要代价的。系统调优后对某些复杂查询性能良好，可能需要付出的代价是，其他查询也许得不到同样的性能提升。

463 要在 InnoDB 中使用 `PERFORMANCE_SCHEMA`，必须要有 MySQL 5.5 或更新版本和 InnoDB 1.1 或更新版本，并且要在服务器上启用 `PERFORMANCE_SCHEMA`。所有的 InnoDB 实例、对象、消费者（consumers）等，名字中都有“innodb”前缀。例如，下面是一个活动的 InnoDB 线程列表（这个可以用来隔离和监控 InnoDB 的执行情况）：

```
mysql> SELECT thread_id, name, type FROM threads WHERE NAME LIKE '%innodb%';
```

thread_id	name	type
2	thread/innodb/io_handler_thread	BACKGROUND
3	thread/innodb/io_handler_thread	BACKGROUND
4	thread/innodb/io_handler_thread	BACKGROUND



```

|      5 | thread/innodb/io_handler_thread      | BACKGROUND |
|      6 | thread/innodb/io_handler_thread      | BACKGROUND |
|      7 | thread/innodb/io_handler_thread      | BACKGROUND |
|      8 | thread/innodb/io_handler_thread      | BACKGROUND |
|      9 | thread/innodb/io_handler_thread      | BACKGROUND |
|     10 | thread/innodb/io_handler_thread      | BACKGROUND |
|     11 | thread/innodb/io_handler_thread      | BACKGROUND |
|     13 | thread/innodb/srv_lock_timeout_thread | BACKGROUND |
|     14 | thread/innodb/srv_error_monitor_thread | BACKGROUND |
|     15 | thread/innodb/srv_monitor_thread     | BACKGROUND |
|     16 | thread/innodb/srv_purge_thread       | BACKGROUND |
|     17 | thread/innodb/srv_master_thread      | BACKGROUND |
|     18 | thread/innodb/page_cleaner_thread    | BACKGROUND |
+-----+-----+-----+
16 rows in set (0.00 sec)

```

我们看到一些 InnoDB 特有的项同时存在于 `rwlock_instances`、`mutex_instances`、`file_instances`、`file_summary_by_event_name` 和 `file_summary_by_instances` 表中。

## 其他需要考虑的参数

监控和优化 InnoDB 存储引擎涉及很多东西。前面我们只讨论了其中的一部分，主要介绍了监控各种子系统和提升性能。当然，还有一些其他参数可能需要考虑。

在某些情况下，调节 `innodb_thread_concurrency` 选项可以提高线程的性能。在 MySQL 5.5 及之后的版本，该参数的默认值是 0（之前的版本是 8），即存储引擎是无限并发的或者运行了很多线程。一般情况下，该值是足够的。但如果 MySQL 服务器上有很多处理器而且有很多独立的磁盘（并频繁使用 InnoDB），那么将此值设置为处理器个数加上独立磁盘数的和，可以提高系统性能。这就保证了 InnoDB 使用足够的线程最大化并发操作。如果把这个值设置得超出了服务器所能支持的范围，则起不到任何作用或作用不大——如果没有任何可用的线程，这个值永远不可达到。

464

如果 MySQL 服务器所在的系统频繁或甚至定期关机（例如，在装有 Linux 系统的笔记本电脑上运行 MySQL），你会注意到关闭 InnoDB 可能需要花费很长时间。幸运的是，设置 `innodb_fast_shutdown` 选项能够快速关闭 InnoDB。这不会影响数据的完整性，也不会给内存（缓冲）管理带来损失。快速关闭 InnoDB 只是简单地跳过了清除内部缓存和合并插入缓冲区这些潜在高成本的操作，这依然是一个可控的关闭过程，并在磁盘上存储缓冲池。

设置 `innodb_lock_wait_timeout` 变量可以控制 InnoDB 如何处理死锁。该变量有全局和会话两个级别，控制 InnoDB 事务在终止之前等待行锁的时间，默认值是 50 秒。如果有

很多锁等待超时，可以减少该变量值降低等待锁的时间。这样有助于诊断某些并发问题，或者至少可以使查询更早超时。

如果你正在导入大量数据，请确保这些数据文件是按照主键顺序导入的，这样可以改善装载时间。此外，将 `AUTOCOMMIT` 设为 0 可关闭自动提交，保证整个装载只提交一次。还可以关闭外键和唯一约束来改善批量装载。



记住，应该非常谨慎地优化 InnoDB。InnoDB 优化过程中有很多东西需要调整，很容易出错。一定要遵循以下原则：一次只调整一个参数（仅为了一个目的），然后不停地测试。

## InnoDB 故障排除的技巧

上面列出的工具都是最佳使用工具，包括 InnoDB 监视器，`SHOW ENGINE INNODB STATUS` 命令（另一种显示 InnoDB 监视器数据的方法），以及 `PERFORMANCE_SCHEMA`。不过，使用 InnoDB 处理错误的时候还有几个有用的策略。这一节介绍 InnoDB 故障排除的一些最佳实践。使用这些实践可处理错误、警告和数据崩溃问题。

### 错误

如果遇到与 InnoDB 相关的错误，去错误日志寻找错误信息。使用 `--log-error` 启动选项开启错误日志。

465

### 死锁

如果遇到多个死锁失败，使用 `--innodb_print_all_deadlocks` 选项（5.5 及之后的版本才有）将所有死锁信息写入错误日志。这样，`SHOW ENGINE INNODB STATUS` 显示的就不仅是最近一个死锁，如果应用程序本身没有死锁错误处理机制的话，这个信息更加有用。

### 数据字典问题

表的定义存储在 `.frm` 文件中，这些文件名与表名相同，位于与数据库名同名的目录下。表定义还会存储在 InnoDB 的数据字典中。如果发生存储崩溃或文件损坏，可能会遇到数据字典不匹配的问题。这里列出了一些常见的症状和解决办法。

#### 孤立的临时表

如果 `ALTER TABLE` 操作失败，服务器可能没有正确清理临时表，还有些临时表留在 InnoDB 表空间中。如果出现这种情况，使用表监视器确定表名（临时表的名字以 `#sql` 开头）。然后使用 `DROP TABLE` 命令删除这个孤表。

## 无法打开表

如果看到这样的错误信息 `Can't open file: 'somename.innodb'`, 错误日志中有这样的信息 `Cannot find table somedb/somename...`, 说明数据库文件夹中有一个名为 `somename.frm` 的孤立文件。这时, 删除这个孤立的 `.frm` 文件就可以了。

## 表空间不存在

如果使用了 `--innodb_file_per_table` 选项, 遇到类似这样的问题 `InnoDB data dictionary has tablespace id N, but tablespace with the id or name does not exist...`, 那么必须删除表然后重新创建它。但是, 可没有那么简单。首先必须在另一个数据库中重建这个表, 找到 `.frm` 文件, 将它复制到原始数据库, 然后删除表。这样可能会产生一个警告信息说找不到 `.ibd` 文件, 但是能够更正数据字典。这样, 就可以重建表并从备份中恢复数据。

## 无法创建表

如果错误日志中有个错误是表已经存在于数据字典, 那么这个表可能没有相应的 `.frm` 文件。如果发生这种情况, 根据错误日志的提示进行处理。

## 观察控制台消息

466

有些错误和警告只有控制台才有 (例如 `stdout`、`stderr`)。在进行错误和警告的故障排除时, 有时候最好通过命令行启动 MySQL, 而不要用 `mysqld_safe` 脚本。在 Windows 中, 使用 `--console` 选项防止控制台消息被禁用。

## I/O 问题

I/O 问题通常在启动的时候或者创建 / 删除新对象的时候出现。这种类型的错误与 InnoDB 文件有关, 严重程度不一。不幸的是, 这些问题通常与平台和操作系统有关, 因此需要特定的解决办法。通常的做法是, 检查错误日志或控制台的错误信息, 检查操作系统相关的错误, 因为这些错误会提示发生 I/O 错误的原因。此外, 还要检查数据目录下丢失或崩溃的文件夹和 InnoDB 文件。

如果数据磁盘出现问题, 也会发生 I/O 问题。通常发生在启动时, 不过任何时候只要出现磁盘读写错误都会出现这个问题。有时候, 硬件错误会被误认为是性能问题。所以, 在做故障排除的时候, 要检查操作系统的磁盘诊断信息。

有时候可能是配置问题导致的。这时, 应该再次检查配置文件, 保证 InnoDB 配置正确。例如, 检查 `innodb_data_*` 选项配置是否正确。



## 数据库崩溃

如果数据库遇到了严重或重大错误，导致 InnoDB 崩溃或者服务器不运行了，在配置文件中使用 `innodb_force_recovery` 恢复选项启动服务器，将其设置为 1 到 6 的整型值，可使 InnoDB 在启动过程中跳过某些操作。



这个选项是终极办法，应该只在某些极端的情况下才使用，比如其他办法启动服务器都失败的时候。而且，在做这个操作之前，还要先把数据导出来。

每个选项的简要描述如下（更多信息参见在线参考手册）：

1. 发出 `select` 语句的时候，跳过损坏的页。仅允许部分数据恢复。
2. 不要启动 `master` 或清除线程。
3. 不要在崩溃恢复之后执行回滚。
- 467 4. 不要执行插入缓冲区操作。不要计算表的统计信息。
5. 启动时忽略撤销（`undo`）日志。
6. 运行恢复时不要执行重做（`redo`）日志。

## MyISAM

MyISAM 存储引擎需要监控的东西很少，这是因为 MyISAM 存储引擎是为 Web 应用程序而构建的，致力于快速查询，因此，拥有该存储引擎的服务器只需要调整 `key cache` 就可以了。这并不是说就没有别的办法改善性能了，相反，有很多事情可以做，比如使用低优先级和并发插入这样的选项。提高性能的方法大致可分为三类：优化磁盘存储，通过监控和优化 `key cache` 来有效地使用内存，以及优化数据库表。

我们并不对每个方面进行深入探讨，而是战略性地将性能提升分成以下几个方面：

- 优化磁盘存储
- 优化数据库表的性能
- 使用 MyISAM 实用工具
- 按照索引顺序存储表
- 压缩表
- 对数据表进行碎片整理



- 监控 key cache
- 预加载 key cache
- 使用多个 key cache
- 其他需要考虑的参数

下面我们逐个进行简单介绍。

## 优化磁盘存储

MyISAM 的磁盘空间优化更像是系统配置项，而不是 MyISAM 的调优参数。MyISAM 将数据表保存为 *.myd*（数据文件）和一个或多个 *.myi*（索引）文件。这些文件与 *.frm* 文件一起存储在与数据库名同名的目录下，由 `--datadir` 启动选项指定。因此，MyISAM 的磁盘空间优化与服务器上的磁盘空间优化方法相同。也就是说，将数据目录移到自己的磁盘上可以提高性能，还可以使用 RAID 或其他高可用性存储选项来进一步提高性能。

◀ 468



最新发布的 MySQL 实用工具中有一个名为 *.frm* 阅读器 (*mysqlfrm*) 的新工具，它可以读取 *.frm* 文件然后生成表的 CREATE 语句。任何需要诊断 *.frm* 文件问题的时候都可以使用这个工具。更多信息请参见 MySQL 实用工具文档。

## 修复表

可以使用以下几种 SQL 命令优化数据库表：ANALYZE TABLE、OPTIMIZE TABLE 和 REPAIR TABLE。

ANALYZE TABLE 命令用于检测和重组表的关键字分布。如果 JOIN 操作使用的是字段而不是常量，MySQL 使用关键字分布决定 JOIN 操作的顺序。关键字分布还决定了查询时使用哪个索引。我们在第 11 章的“使用 ANALYZE TABLE”一节中已经详细介绍了这个命令。

REPAIR TABLE 命令并不算一个真正的性能工具，其用于为 MyISAM、Archive 和 CVS 存储引擎修复崩溃的表。该命令用于恢复那些崩溃的或运行很慢的表（这通常表明该表已经退化，需要重组或修复）。

OPTIMIZE TABLE 命令用于恢复被删除的块和重组表，从而提高数据库性能。可以在 MyISAM 和 InnoDB 表上使用该命令。

尽管这些命令很实用，但还有很多高级工具可以用于管理 MyISAM 表。

## 使用 MyISAM 实用工具

MySQL 发布包中包含了很多管理 MyISAM 存储引擎（表）的专用工具。

- *myisam\_ftdump*：显示全文索引信息。
- *myisamchk*：在 MyISAM 表上执行分析。
- *myisamlog*：查看 MyISAM 表的更改日志。
- *myisampack*：压缩表以减少存储量。

469 *myisamchk* 是监控 MyISAM 的主力工具，能够显示 MyISAM 表的信息，或者对这些表进行分析、修复和优化。可以在一个或多个表上运行该命令，但是如果需要刷新和锁定表的话，只能在服务器运行的时候才能使用。否则，可以关闭服务器。



在运行该工具前，请务必备份表，防止修复或优化失败。这样可以避免表损坏或不可修复，虽然这种情况极少发生。

下面列出了关于性能提升、恢复和状态报告的选项（这些选项的完整说明请参看在线 MySQL 参考手册）：

### analyze

分析索引的关键字分布以提高查询性能。

### backup

在更改表之前复制一份副本（即 *.myd* 文件）。

### check

检查表的错误信息（仅报告）。

### extended-check

彻底检查表（包括索引）的错误信息（仅报告）。

### force

如果发现错误，则执行修复。

### information

显示表的统计信息。在运行 *recover* 命令恢复表之前，使用该命令查看表的状况。

### medium-check

更加深入地检查表（仅修复）。这比 *extended-check* 检查的信息少。

recover

全面修复表（修复数据结构）。执行除了唯一键重复的所有修复工作。

safe-recover

执行传统形式的修复，即有序地读取所有行，并更新所有索引。

sort-index

从高到低排列索引树。这样能够减少索引结构的查找时间，加快索引的访问速度。

sort records

按指定的索引顺序对记录进行排序。这样可以提高某些基于索引的查询的性能。

示例 12-6 展示了运行 `myisamchk` 命令后显示的 MyISAM 表信息。

示例12-6: `myisamchk` 工具

```
MyISAM file:      /usr/local/mysql/data/employees/employees
Record format:    Packed
Character set:    latin1_swedish_ci (8)
File-version:     1
Creation time:    2012-12-03 08:13:03
Status:          changed
Data records:     297024 Deleted blocks:      3000
Datafile parts:   300024 Deleted data:        95712
Datafile pointer (bytes): 6 Keyfile pointer (bytes): 6
Datafile length:  9561268 Keyfile length:     3086336
Max datafile length: 281474976710654
Max keyfile length: 288230376151710719
Recordlength:    44
table description:
Key Start Len Index Type      Rec/key      Root      Blocksize
1  1      4  unique long          1      2931712      1024
```

## 按索引顺序存储表

按照索引顺序存储表数据可以提高大量的数据范围查询（例如 `WHERE a > 5 AND a < 15`）的检索效率。这种排序允许查询有序地访问数据，而无须查找磁盘页。为了按照索引顺序对表进行排序，可以使用 `myisamchk` 工具的排序记录选项（`-R`），并指定使用哪个索引，索引编号从 1 开始。下面的命令按照第二个索引顺序对 `test` 数据库中的 `table1` 表进行排序：

```
myisamchk -R 2 /usr/local/mysql/data/test/table1
```

使用 `ALTER TABLE` 和 `ORDER BY` 命令可以达到同样的效果。

当在数据表中添加新行时，这样排序表的方式无法保证表数据仍按索引顺序存储。删除操作不影响排序，但是添加新行会使表不再有序，导致数据库性能下降。如果在经常变更的表上采用这个技术，可能需要定期运行该命令以确保表的存储顺序最佳。

471

## 压缩表

压缩数据可以节约空间。虽然 MySQL 中压缩数据的方法很多，但是 MyISAM 存储引擎只能压缩 (pack) 只读表，因为 MyISAM 不能解压、重新排序，也不能对压缩数据执行添加（或删除）操作。MyISAM 中使用 *myisampack* 压缩表，如下所示：

```
myisampack -b /usr/local/mysql/data/test/table1
```

在压缩数据表前，使用备份选项 (-b) 创建表的备份副本。这样可以在不需要重新运行 *myisampack* 命令的情况下，使表变为可写的。

压缩只读表有两个原因。首先，那些易于压缩的数据（如文本）表可以节约存储空间。其次，当查询读取压缩后的表，并通过主键或唯一索引来查找表中的某一行时，在比较其他条件之前，仅对单行数据进行解压缩。

*myisampack* 命令有许多选项。如果你对压缩只读表有兴趣，请参阅在线 MySQL 参考手册，以了解怎样操控压缩功能的细节。

## 对数据表进行碎片整理

当对 MyISAM 数据表有很多删除和插入等变更操作时，物理存储将变得很零散。通常，已删除的数据给物理存储带来小间隙，或者原来的存储顺序被破坏，或者同时出现这两种情况。为了优化数据表，将其重新组织成期望的顺序和形式，可以使用 *OPTIMIZE TABLE* 命令或 *myisamchk* 实用工具。

如果需要指定特定的排序顺序，应该定期运行这些命令，以确保这些表的存储形式最优。另外，如果某段时间内数据经历了多次更新，也应该运行这些命令。

## 监控 key cache

MySQL 的 key cache 是一个高效的存储结构，用于存储频繁使用的索引数据。只有 MyISAM 才能使用 key cache，通过快速查找机制（通常是 B-tree）存储关键字。索引的内部存储形式是链接列表（存储在内存中），可以被快速检索到。在打开第一个 MyISAM 数据表读取时自动创建 key cache。每次查询 MyISAM 数据表前，都会检查一遍 key cache。如果有索引，则直接在内存中执行索引检索，而不需要先从磁盘上读取索引。key cache 是使 MyISAM 的快速查询比其他存储引擎都快的秘密武器。



有许多变量用于控制 key cache，可以使用 SHOW VARIABLES 和 SHOW STATUS 命令监控这些变量。示例 12-7 显示了用 SHOW 命令监控这些变量的实例。

示例12-7: key cache的状态和系统变量

```
mysql> SHOW STATUS LIKE 'Key%';
```

Variable_name	Value
Key_blocks_not_flushed	0
Key_blocks_unused	6694
Key_blocks_used	0
Key_read_requests	0
Key_reads	0
Key_write_requests	0
Key_writes	0

7 rows in set (0.00 sec)

```
mysql> SHOW VARIABLES LIKE 'key%';
```

Variable_name	Value
key_buffer_size	8384512
key_cache_age_threshold	300
key_cache_block_size	1024
key_cache_division_limit	100

4 rows in set (0.01 sec)

可以想象，key cache 机制非常复杂。所以 key cache 调优将是一个挑战。建议监控 key cache 的使用情况，并根据情况改变 key cache 的大小，最好不要改变它的运行方式，因为默认配置已能够满足运行要求。

如果想提高缓存命中率，可以使用以下两种方法：(1) 预加载缓存；(2) 使用多个 key cache 并为默认 key cache 分配更多的内存。下面我们分别讨论这两种技术。

## 预加载 key cache

将索引预加载到 key cache 可以加快查询速度，因为索引已经加载到缓存，并且是按顺序加载的（而不是随机的，例如并发操作下的 key cache 就是随机加载的）。然而，必须保证缓存中有足够的空间存放索引。对某些特定应用或使用模型来说，预加载是提高查询速度的有效方法。比如，如果在应用程序（例如典型的工资审计程序）的执行过程中，某个特定的表被查询很多次，这时你可以将该表的相关索引预加载到 key cache 中，从

而提高性能。使用 LOAD INDEX 命令可执行预加载，如示例 12-8 所示。

示例12-8：将索引预加载到key cache

```
mysql> LOAD INDEX INTO CACHE salaries IGNORE LEAVES;
```

Table	Op	Msg_type	Msg_text
employees.salaries	preload_keys	status	OK

1 row in set (1.49 sec)

这个例子将 salary 表的索引加载到 key cache。IGNORE LEAVES 从句表明只预加载索引的非叶子节点。虽然没有特殊的命令用于刷新 key cache，但是可以通过修改表强行从 key cache 中删除索引，例如重组索引，或删除并重建索引。

## 使用多个 key cache

MySQL 有一个鲜为人知的高级特性，即创建多个 key cache 或自定义 key cache，以减少对默认 key cache 的争用。该特性允许将一个或多个表的索引加载到某个特殊的缓存中。这意味着按任务分配内存，需要认真规划。如果某段时间内对一组表执行大量查询操作，而且频繁引用这些表上的索引，那么这个方法则能够大大提高性能。

要创建一个二级 key cache，首先需要使用 SET 命令分配内存，然后执行一个或多个 CACHE INDEX 命令加载一个或多个表的索引。与默认 key cache 不同的是，可以通过将二级 key cache 的大小设为 0 将其刷新或删除。示例 12-9 显示了如何创建二级 key cache，然后将表的索引添加到缓存中。

示例12-9：使用二级key cache

```
mysql> SET GLOBAL emp_cache.key_buffer_size=128*1024;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CACHE INDEX salaries IN emp_cache;
```

Table	Op	Msg_type	Msg_text
employees.salaries	assign_to_keycache	status	OK

1 row in set (0.00 sec)

474

```
mysql> SET GLOBAL emp_cache.key_buffer_size=0;
```

```
Query OK, 0 rows affected (0.00 sec)
```

注意，二级缓存需要定义一个名为 emp\_cache 的新变量，并将其大小设置为 128KB。这

是 SET 命令的特殊语法，虽然看起来像是创建新的系统变量，但其实是创建了一个新的全局用户变量。通过以下方式可确定二级 key cache 是否存在或其大小：

```
mysql> select @@global.emp_cache.key_buffer_size;
+-----+
| @@global.emp_cache.key_buffer_size |
+-----+
|                                131072 |
+-----+
1 row in set (0.00 sec)
```

由于二级 key cache 是全局的，因此除非你将其大小设置为 0 刷新了它，或者重启了服务器，否则二级 key cache 一直存在。



保存多个 key cache 配置的方法是：将配置语句保存在一个文件中，然后使用 `init-file=path_to_file` 命令在 MySQL 选项文件的 [mysql] 部分设置这个文件，系统启动的时候会执行这些语句。

## 其他需要考虑的参数

还有很多其他参数需要考虑。请记住，一次只能改变一个参数，并且只有当理由充分时才需要改变参数。在没有好的理由和合理的期望结果的情况下，永远不要更改复杂项的配置，如存储引擎。

### myisam\_data\_pointer\_size

如果没有为 MAX\_ROWS（表中存储的最大行数）指定值，CREATE TABLE 使用的默认指针大小一般取 2~7（单位为字节）。默认值为 6。

### myisam\_max\_sort\_file\_size

数据排序时使用的临时文件大小的最大值。增大该值可以加速索引的修复和重组。

### myisam\_recover\_options

MyISAM 的恢复模式。也可用于 OPTIMIZE TABLE。该模式包括默认（default）、备份（backup）、强制（force）、快速（quick），这些选项可以任意组合。默认模式是指在不检查备份、强制或快速的情况下执行恢复。备份模式是指在恢复前首先创建备份。强制模式是指即使数据丢失（不止一行），仍进行数据恢复。快速模式是指如果没有标记为删除的块，就不检查表中的数据行。根据恢复的严重性决定使用哪些项。

475

`myisam_repair_threads`

如果该值大于 1，则并行执行修复和排序操作，从而加快操作速度。否则，这些操作将顺序执行。

`myisam_sort_buffer_size`

排序操作的缓存区大小。增加该值有助于排序索引，但是如果该值超过 4GB，只适用于 64 位的机器。

`myisam_stats_method`

在统计操作中用于控制服务器如何统计索引值分布中的 NULL 值。这会影响到优化器，请小心使用。

`myisam_use mmap`

为读写 MyISAM 表开启存储器映射（memory map）选项。如果同时存在很多小写入和返回大数据集的读查询，这个功能非常有用。

我们已经讨论了很多关于监控和改进 MyISAM 性能的策略。虽然比较简单，但是覆盖了高效使用 MyISAM 最重要的部分。请参看在线 MySQL 参考手册，以获得关于 key cache 和 MyISAM 存储引擎的更多信息。

## MySQL、复制和高可用性

MyISAM 数据损坏的概率比 InnoDB 高，因此，MyISAM 需要较长的恢复时间。此外，由于 MyISAM 不支持事务，一次只能执行一个事件，这会导致语句的部分执行，因此会出现不完整的事务。另外，由于 slave 是单线程执行的，在处理长时间运行的查询时，slave 可能会滞后。因此，在一个包含事务的高可用性解决方案中，在 slave 上使用 MyISAM 将会出问题。

## 小结

本章研究了如何监控和提高 MySQL 服务器中存储引擎的性能。我们详细讨论了两种最流行的存储引擎，下一章将讨论更高级的主题：监控和提高复制的性能。

476

Joel 将鼠标停留在发送键上。他刚刚做完一份 InnoDB 监控数据的报告，并在邮件里写了一些建议。但是他不知道是否应该在老板还没要之前就发给他。他耸了耸肩，心想反正也没什么坏处，就点击了发送。

大约两分钟后，系统弹出一个消息框提示有新邮件。Joel 打开邮件，是老板发过来的。



“Joel, InnoDB 的事情做得不错。我希望你能够跟开发人员和 IT 人员一起开个会,大家坐下来谈谈你的建议。你来安排吧,我星期一来公司。”

“好的。”Joel 说,感觉肩上多了份责任感。这可是他上班以来的第一次会议。他有些紧张,所以决定出去散散步,然后再把相关议程发送给需要参加会议的人。“嗯,这肯定不会比我的论文答辩更糟糕吧。”

# 监控复制

Joel 花了点时间登录西雅图的复制 slave，确定复制仍在运行。

一个熟悉的声音从门口传来：“西雅图那边怎么样了，Joel，你在跟进这件事情吗？”

“我正在处理呢，先生。我需要找出复制的配置信息，然后监控问题所在。”Joel 心想，还要阅读更多关于监控复制的知识。

“好的，Joel。继续工作吧，午饭后我再来。”

当老板离开后，Joel 看了看表，“好吧，我还有大约一个小时的时间想出如何监控复制。”

Joel 深深地叹了一口气，再次打开自己喜爱的 MySQL 书，学习更多有关 MySQL 监控的知识。“没想到复制本身会带来这么多问题，”他喃喃地说。

你已经知道服务器什么时候运行良好（以及什么时候不好），那么如何知道复制的运行情况呢？也许很顺利，但是你怎么知道呢？

这一章讨论高级监控，重点介绍监控和提高复制的性能。

## 入门

有两个方面影响复制拓扑性能。必须同时优化它们，以免影响复制。

首先，确保网络有足够的带宽去处理复制数据。我们讲过，master 产生更新的副本，然后通过网络将副本发送给 slave。如果网络连接速度很慢或者资源竞争很激烈，那么复制数据也会很慢。我们将介绍一些优化网络和复制的方法，以最大限度地利用当前的网络环境。

其次，也是最重要的，要确保被复制的数据库是优化过的。这一点很重要，因为 master 数据库的任何低效率的事件都会被同步到 slave 上，从而导致 slave 的数据库性能也不高，特别是索引和规范化。然而，优化数据库只是优化复制的一部分，还必须确保查询语句是优化的。例如，性能不好的查询在 master 上运行缓慢，也会在 slave 上运行不佳。

一旦网络好了，数据库和查询都优化过了，就可以专注于配置服务器以获得最优性能了。

## 服务器设置

为复制拓扑打造最佳平台还有一件很重要的事情，就是确保服务器的配置是性能最优的。性能不好的复制拓扑往往与性能差的服务器有关。确保服务器操作系统拥有足够内存，而且存储设备和存储引擎对数据库来说都是最优的。

有人建议 slave 使用低性能的机器，因为 slave 上的运行负载较少（一般来说，slave 只处理 SELECT 查询，而 master 需要处理数据更新）。但这是不对的。对于典型的单 master 和单 slave 环境，所有的数据库都需要复制，两个机器的负载几乎相同，但是 slave 使用单线程执行事件，而 master 使用多线程执行事件，所以即使负载相同，slave 处理和执行事件所花的时间可能更多。

对此，最好的办法是考虑复制的最佳用途之一——故障转移。如果 slave 比 master 慢，并且如果 master 出现故障就必须进行故障转移，期望的结果是提升后的 slave 应该与崩溃前的 master 拥有相同的性能。

## 包容性和排他性复制

你可以将复制配置成复制所有数据（这是默认配置）；也可以只记录 master 上的某些数据或者忽略某些数据，从而限定哪些写入二进制日志，哪些被复制；或者还可以配置 slave，使其只对某些数据进行操作。使用包容性或排他性复制（或同时使用两种复制）有助于解决复杂的负载均衡或系统扩展问题，使复制更强大、更灵活。这个过程又称为数据过滤，其中包容性和排他性要求形成了过滤规则。

◀ 479

在 master 上使用 `--binlog-do-db` 启动选项指定只将某个数据库的事件写入二进制日志。可以在命令行或配置文件中指定多个该选项，每个选项指定一个数据库。

还可以使用 `--binlog-ignore-db` 启动选项指定忽略某个数据库的事件，可以在命令行或配置文件中指定多个该选项，每个选项指定一个数据库。该选项告诉 master 这些数据库的事件不需要写入日志。



如果指定的数据库不重复的话，可以同时使用 `--binlog-do-db` 和 `--binlog-ignore-db` 选项。如果某个数据库名同时出现在这两个选项中，那么 `--binlog-ignore-db` 中的那个数据库就被忽略。确保在诊断数据复制问题（例如 slave 数据丢失）的时候检查这些变量值。

另外，使用 `-binlog-do-db` 或 `-binlog-ignore-db` 选项会过滤写入二进制日志的数据。这严重限制了 PITR 的使用，因为只有写入二进制日志的数据才能恢复。

还有几个选项用来控制哪些数据需要复制到 slave，包括 master 上的 binlog 选项，限制表级别的选项，甚至还有做转换（重命名）的命令选项。

在 slave 上执行包容性或排他性复制或许无法提高拓扑复制的性能。尽管 slave 上的数据较少，master 还是需要传输相同的数据量，而且如果包容性和排他性列表很复杂，那么在 slave 上执行过滤带来的开销并没有什么帮助。如果你担心通过网络传输的数据太多，最好在 master 上执行过滤操作。

使用 `--replicate-do-db` 启动选项可指定 slave 数据库上的哪些事件可以从中继日志中读取然后执行。可以在命令行或配置文件中指定一个或多个这个选项，每个选项指定一个数据库。该选项告诉 slave 只执行指定数据库上的事件。

使用 `--replicate-ignore-db` 启动选项指定哪些数据库上的事件将被忽略。可以在命令行或配置文件中指定一个或多个这个选项，每个选项指定一个数据库。该选项告诉 slave 忽略指定数据库上的所有事件。

480



slave 上的复制选项根据使用格式不同而作用不同。对于基于语句的复制而言，这一点尤为重要，稍有不慎就可能导致数据丢失。例如，如果使用基于语句的复制，且使用 `--replicate-do-db` 选项，那么 slave 只对 `USE <db>` 命令后的那些语句的事件起作用。如果你在不改变数据库的情况下向另一个数据库发送语句，该语句将会被忽略。想要了解更多信息，请参看在线 MySQL 参考文档。

可以在 slave 上执行表级别的包容性和排他性复制。使用 `--replicate-do-table` 和 `--replicate-ignore-table` 选项可执行或忽略特定表的事件。如果表总包含敏感数据，这些表用于管理或某些特殊用途而应用程序不可使用，上述命令是非常有用的。例如，如果有一个应用程序包含供应商的定价信息（你付出的钱），在雇佣或外包销售服务时你可能想要隐藏这些信息。在这种情况下，不需要单独为承包商创建一个特殊的应用，而是部署现有的应用程序，使用 slave 复制除了带有敏感数据的表的所有信息。

上面两个选项还有通配符形式，即 `replicate-wild-do-table` 和 `replicate-wild-ignore-table`，它们与前面两个选项功能相同，还支持通配符。例如，`--replicate-`



wild-do-table=db1.tbl% 执行数据库 *db1* 中任何以“tbl”开头的所有表（如 tbl、tbl1、tbl\_test）的事件。同样，使用 --replicate-wild-do-table=db1.% 将执行数据库 *db1* 中的所有表的事件。这些通配符在 slave 端过滤，有利于解决复杂复制问题。

还可以在 slave 上使用转换选项，用于重命名或更改表操作对应的数据库。该选项仅适用于表，格式为 --replicate-rewrite-db="<from>-><to>"（必须加引号）。该选项只改变表事件对应的数据库名，不影响 CREATE DATABASE、ALTER DATABASE 等命令。该选项只影响指定数据库的事件（或者为基于语句的复制重定向默认数据库）。可以多次使用该选项改变多个数据库的名称。



虽然 --replicate-same-server-id 不是严格意义上的包容性或排他性选项，但可以防止在循环复制中产生无限循环。如果将其值设置为 0，slave 将跳过相同 server\_id 的事件；如果将其值设置为 1，slave 将执行所有事件。

## 复制线程

◀ 481

在谈论 master 和 slave 的监控问题之前，首先应该复习一下复制中的线程问题。这里我们再次从监控和诊断问题的角度讨论一下。

有三个线程控制复制，每个线程执行一个特定的角色。在 master 上，每个连接的 slave 都有一个线程，称为 Binlog Dump 线程。该线程负责将 binlog 事件传送给连接的 slave。slave 上有两个线程，即 Slave IO 线程和 Slave SQL 线程。I/O 线程负责读取 master 发过来的 binlog 事件，然后将这些事件写入 slave 的中继日志。SQL 线程负责读取并执行中继日志中的事件。

使用 SHOW PROCESS LIST 命令可监控 Binlog Dump 线程的当前状态：

```
mysql> SHOW PROCESSLIST \G
***** 1. row *****
   Id: 1
  User: rpl
  Host: localhost:54197
   db: NULL
Command: Binlog Dump
   Time: 25
  State: Master has sent all binlog to slave; waiting for binlog to be updated
   Info: NULL
```

请注意 State 字段，描述 master 对二进制日志和 slave 做了哪些操作。这个例子显示了一个运行良好的复制拓扑产生的典型结果。其中还包含以下字段：

**Id**  
显示连接 ID。

**User**  
显示运行该语句的用户。

**Host**  
语句来自哪个主机。

**db**  
指定的默认数据库；如果未指定，则显示 NULL，表明没有指定默认数据库。

**Command**  
该线程运行的命令类型。参见在线 MySQL 参考手册可获取更多信息。

**Time**  
线程处于报告状态的时间（以秒为单位）。

**State**  
描述当前动作或状态（如等待）。通常是描述性文本信息。

**Info**  
线程正在执行的语句信息。NULL 表明没有语句正在执行，当复制线程处于等待状态时该字段也为 NULL。

此外，还可以在 slave 上查看线程状态。使用 SHOW PROCESSLIST 命令可监控 I/O 和 SQL 线程。

```
mysql> SHOW PROCESSLIST \G
***** 1. row *****
      Id: 2
     User: system user
      Host:
         db: NULL
  Command: Connect
       Time: 127
   State: Waiting for master to send event
      Info: NULL
***** 2. row *****
      Id: 3
     User: system user
      Host:
         db: NULL
```

```
Command: Connect
Time: 10
State: Slave has read all relay log; waiting for the slave I/O thread to
      update it
Info: NULL
```

同样，State字段包含了最重要的信息。如果slave上的复制出了问题，一定要在slave上执行SHOW PROCESSLIST命令，留意I/O和SQL线程的状态。在这个例子中，我们看到了slave处于正常状态，即等待来自master的信息（I/O线程），然后执行中继日志中的所有事件（SQL线程）。



在排除故障时，最好使用SHOW PROCESSLIST命令参看复制状态。

## 监控 master

483

有几种方法可监控master：使用SHOW命令或MySQL工作台查看状态信息和状态变量。主要的SQL命令有：SHOW MASTER STATUS、SHOW BINARY LOGS和SHOW BINLOG EVENTS。

本节我们介绍监控master的SQL命令，然后简单总结一下可监控的状态变量，这些状态变量可以通过SHOW STATUS命令或使用MySQL工作台创建自定义图表进行监控。

### master 的监控命令

SHOW MASTER STATUS命令显示了master的二进制日志的相关信息，包括当前binlog文件名及其偏移位置。这些信息对于连接slave来说很重要，这一点已经在前面的章节中讨论过。此外，这个命令还显示日志约束的相关信息。示例13-1显示了SHOW MASTER STATUS命令的典型结果。

示例13-1：SHOW MASTER STATUS命令

```
mysql> SHOW MASTER STATUS \G
***** 1. row *****
      File: mysql-bin.000002
      Position: 156058362
      Binlog_Do_DB: Inventory
      Binlog_Ignore_DB: Vendor_sales
      Executed_Gtid_Set: 87e02a46-5363-11e2-9d4a-ed25ee3d6542
1 row in set (0.00 sec)
```

显示结果包含以下字段：

File

当前 binlog 文件的名称。

Position

二进制日志的当前位置（即下一次写入的位置）。

Binlog\_Do\_DB

--binlog-do-db 启动选项指定的所有数据库。

Binlog\_Ignore\_DB

--binlog-ignore-db 启动选项指定的所有数据库。

484 Executed\_Gtid\_Set

master 上执行的所有 GTID。只有当服务器启用 GTID 的时候这个字段才有效。这个字段的值与服务器变量 `gtid_executed` 的值一样。

SHOW BINARY LOGS 命令（或 SHOW MASTER LOGS 命令）列出了 master 上可用的二进制文件及其文件大小（以字节为单位）。该命令有利于比较 slave 相对于 master 的信息，例如 slave 目前正在读取 master 上的哪个二进制日志。示例 13-2 显示了 SHOW BINARY LOGS 命令的结果信息。



在 master 上运行 FLUSH LOGS 命令可以轮换二进制日志。该命令关闭并重新打开所有日志，然后以递增的文件扩展名打开一个新日志文件。应该定期刷新日志，有助于管理日益增长的日志，同时还能帮助解决诊断复制的问题。

示例13-2：SHOW MASTER LOGS命令

```
mysql> SHOW MASTER LOGS;
```

```
+-----+-----+
| Log_name          | File_size |
+-----+-----+
| master-bin.000001 | 103648205 |
| master-bin.000002 | 2045693   |
| master-bin.000003 | 1022910   |
| master-bin.000004 | 3068436   |
+-----+-----+
4 rows in set (0.00 sec)
```

使用 SHOW BINLOG EVENTS 命令显示二进制日志中的事件，语法如下：



```
SHOW BINLOG EVENTS [IN <log>] [FROM <pos>] [LIMIT [<offset>,) <rows>]
```

使用这个命令时要小心，因为它会产生大量数据。该命令最好的用处是：将 master 上的事件与 slave 中继日志中的事件进行对比。示例 13-3 显示了典型复制配置中的 binlog 事件。

### 示例13-3：SHOW BINLOG EVENTS命令（基于语句的复制）

```
mysql> SHOW BINLOG EVENTS IN 'master-bin.000001' FROM 2571 LIMIT 4 \G
```

```
***** 1. row *****
```

```
Log_name: master-bin.000001
```

```
Pos: 2571
```

```
Event_type: Query
```

```
Server_id: 1
```

```
End_log_pos: 2968
```

```
Info: use `employees`; CREATE TABLE salaries (
```

```
emp_no      INT      NOT NULL,
```

```
salary      INT      NOT NULL,
```

```
from_date   DATE      NOT NULL,
```

```
to_date      DATE      NOT NULL,
```

```
KEY         (emp_no),
```

```
FOREIGN KEY (emp_no) REFERENCES employees (emp_no)
```

```
ON DELETE CASCADE,
```

```
PRIMARY KEY (emp_no, from_date)
```

```
)
```

```
***** 2. row *****
```

```
Log_name: master-bin.000001
```

```
Pos: 2968
```

```
Event_type: Query
```

```
Server_id: 1
```

```
End_log_pos: 3041
```

```
Info: BEGIN
```

```
***** 3. row *****
```

```
Log_name: master-bin.000001
```

```
Pos: 3041
```

```
Event_type: Query
```

```
Server_id: 1
```

```
End_log_pos: 3348
```

```
Info: use `employees`; INSERT INTO `departments` VALUES
```

```
('d001','Marketing'),('d002','Finance'),('d003','Human Resources'),
```

```
('d004','Production'),('d005','Development'),('d006','Quality
```

```
Management'),('d007','Sales'),('d008','Research'),('d009',
```

```
'Customer Service')
```

```
***** 4. row *****
```

```
Log_name: master-bin.000001
```

```
Pos: 3348
```

```
Event_type: Xid
```

485

```

Server_id: 1
End_log_pos: 3375
Info: COMMIT /* xid=17 */
4 rows in set (0.01 sec)

```

本例使用基于语句的复制。如果使用基于行的复制，binlog 事件的显示结果就很不一样，如示例 13-4 所示。

#### 示例13-4: SHOW BINLOG EVENTS命令（基于行的复制）

```

mysql> SHOW BINLOG EVENTS IN 'master-bin.000001' FROM 2571 LIMIT 4 \G
***** 1. row *****
Log_name: master-bin.000001
Pos: 2571
Event_type: Query
Server_id: 1
End_log_pos: 2968
Info: use `employees`; CREATE TABLE salaries (
  emp_no      INT          NOT NULL,
  salary      INT          NOT NULL,
  from_date   DATE         NOT NULL,
  to_date     DATE         NOT NULL,
  KEY         (emp_no),
  FOREIGN KEY (emp_no) REFERENCES employees (emp_no) ON DELETE CASCADE,
  PRIMARY KEY (emp_no, from_date)
)
***** 2. row *****
Log_name: master-bin.000001
Pos: 2968
Event_type: Query
Server_id: 1
End_log_pos: 3041
Info: BEGIN
***** 3. row *****
Log_name: master-bin.000001
Pos: 3041
Event_type: Table_map
Server_id: 1
End_log_pos: 3101
Info: table_id: 15 (employees.departments)
***** 4. row *****
Log_name: master-bin.000001
Pos: 3101
Event_type: Write_rows
Server_id: 1
End_log_pos: 3292

```

```
Info: table_id: 15 flags: STMT_END_F
4 rows in set (0.01 sec)
```

注意，基于行的情况下二进制日志的信息少很多。当在诊断数据崩溃或间歇性错误等复杂问题时，有时候切换到基于语句的行格式是有好处的。例如，这样将有利于查看写入 master 二进制日志的确切内容，然后将其与 slave 中继日志的读取结果进行比较。如果不一样，基于语句的格式比基于行的格式更容易发现这些不同。第 3 章详细介绍了二进制日志的格式，以及各种格式之间的优缺点比较。



*mysqlbinlog* 工具有个 `--verbose` 选项，能够从行事件构建 SQL 命令。如果想要通过基于行的日志事件发现每个事件的更多信息，就使用这个选项。注意重构的 SQL 语句并不是完整的，从某种意义上说它们等同于原始查询。因此，使用这个选项从行格式转换到 SQL，并不能完全重构事件。

## master 的状态变量

487

监控 master 的状态变量只有两个，即表示 master 上的命令执行次数的计数器：

### Com\_change\_master

显示 `CHANGE MASTER` 命令执行的次数。如果该值变化得很频繁，或者该值比服务器数乘以 slave 的计划重启次数的积高很多，可能是添加的 slave 重启过于频繁，这表明连接不稳定。

### Com\_show\_master\_status

显示 `SHOW MASTER STATUS` 命令执行的次数。与 `Com_change_master` 相同，如果该值很高，表明 slave 的重连接请求次数不正常。

## 监控 slave

有几种方法监控 slave：使用 `SHOW` 命令或 MySQL 工作台查看状态信息和状态变量。主要的 SQL 命令有：`SHOW SLAVE STATUS`、`SHOW BINARY LOGS` 和 `SHOW BINLOG EVENTS`。

本节我们讨论监控 slave 的 SQL 命令，然后简单总结一下可监控的状态变量，这些状态变量可以使用 `SHOW STATUS` 命令，或通过 MySQL 工作台创建自定义图表进行监控。我们将在本章后面的“使用 MySQL 工作台监控复制”一节中介绍 MySQL 工作台。

## slave 的监控命令

SHOW SLAVE STATUS 命令显示以下信息：slave 的二进制日志、slave 到 master 的连接和复制活动，包括当前 binlog 文件的文件名及其偏移位置。从前面章节我们已经知道，这些信息在诊断 slave 性能时非常重要。示例 13-5 显示了在 MySQL 5.6 上执行的 SHOW SLAVE STATUS 命令的典型结果。

示例13-5：SHOW SLAVE STATUS命令

```
mysql> SHOW SLAVE STATUS \G
```

```
***** 1. row *****
Slave_IO_State: Waiting for master to send event
Master_Host: localhost
Master_User: rpl
Master_Port: 3306
Connect_Retry: 60
Master_Log_File: mysql-bin.000002
Read_Master_Log_Pos: 39016226
Relay_Log_File: relay-bin.000004
Relay_Log_Pos: 9353715
Relay_Master_Log_File: mysql-bin.000002
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
Replicate_Do_DB:
Replicate_Ignore_DB:
Replicate_Do_Table:
Replicate_Ignore_Table:
Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
Last_Errno: 0
Last_Error:
Skip_Counter: 0
Exec_Master_Log_Pos: 594
Relay_Log_Space: 1008
Until_Condition: None
Until_Log_File:
Until_Log_Pos: 0
Master_SSL_Allowed: No
Master_SSL_CA_File:
Master_SSL_CA_Path:
Master_SSL_Cert:
Master_SSL_Cipher:
Master_SSL_Key:
Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
```



```

        Last_IO_Errno: 0
        Last_IO_Error:
        Last_SQL_Errno: 0
        Last_SQL_Error:
Replicate_Ignore_Server_Ids:
        Master_Server_Id: 2
            Master_UUID: 87e02a46-5363-11e2-9d4a-ed25ee3d6542
        Master_Info_File: /Users/cbell/source/temp_13002/master.info
            SQL_Delay: 0
        SQL_Remaining_Delay: NULL
        Slave_SQL_Running_State: Slave has read all relay log; waiting for the slave
                                I/O thread to update it
        Master_Retry_Count: 86400
            Master_Bind:
        Last_IO_Error_Timestamp:
        Last_SQL_Error_Timestamp:
            Master_SSL_Crl:
        Master_SSL_Crlpath:
        Retrieved_Gtid_Set: 87e02a46-5363-11e2-9d4a-ed25ee3d6542:1-2
        Executed_Gtid_Set: 87e02a46-5363-11e2-9d4a-ed25ee3d6542:1-2,
                        d28c2ea6-5362-11e2-9d45-c78c6761ae47:1
        Auto_Position: 1
1 row in set (0.00 sec)

```

以上例子显示了大量信息。该命令是有关复制的最重要的命令，最好能仔细研究上面例子中的每一项。这里我们就不逐项列出了，而是从管理员的角度解释这些信息。也就是说，通常这些信息是有明确目的的。因此，为了方便参看，我们将这些信息进行分组，包括：master 连接信息、slave 性能、日志信息、过滤、日志性能和错误条件。

◀ 489

第一行信息最重要，显示了当前 I/O 线程的状态。这个状态可能是：正在连接 master、等待 master 的事件、重新连接 master 等。

master 连接的信息包括当前 master 的主机名、连接的用户账号，以及用于连接 master 的 slave 端口。最后是 SSL 连接信息（如果使用了 SSL 连接的话）。

下一类是 master 二进制日志和 slave 中继日志的信息，包括文件名和位置信息。诊断复制问题时，必须留意这些值。尤其是 Relay\_Master\_Log\_File 的值，该值表明了中继日志最近事件所在的 master 日志文件名。

复制过滤配置列举了所有 slave 端的复制过滤器。如果不知道过滤器的配置，可以检查这个配置。

另外，还有 slave 和 I/O、SQL 线程的最近的错误号和文本。除了 slave 线程的状态值，

如果出现错误常常需要检查这些信息。当 slave 遇到错误时，在检查错误日志之前，最好先查看这些信息，因为这些信息最及时，通常可以告诉你出错的原因。

还有 slave 的配置信息，包括跳过计数器的设置和 until 条件。参看在线 MySQL 参考手册可获取更多信息。

这个列表底部显示的是当前的错误信息，包括 slave 的 I/O 和 SQL 线程的错误信息。如果 slave 正常运行，这些值应该总是为 0。

下面我们详细讨论一些较重要的性能字段。

#### Connect\_Retry

两次重新连接的时间间隔（以秒计算）。该值应该较小，但是如果 slave 连接 master 时出现问题，可以将该值设置得较大。

#### 490 Exec\_Master\_Log\_Pos

显示 master 二进制日志中最后执行的事件位置。

#### Relay\_Log\_Space

所有中继日志文件的总大小。根据事件运行过程中的磁盘空间状况，确定是否需要清除中继日志。

#### Seconds\_Behind\_Master

事件执行和事件写入 master 二进制日志之间的间隔时间（以秒计算）。该值过高表明复制有重大滞后。我们马上讨论复制滞后。



当复制由于网络错误、master 的心跳丢失等原因停止时，Seconds\_Behind\_Master 的值将过期。这个值只有在复制运行时才有效。

#### Retrieved\_Gtid\_Set

slave 接收到的 GTID（事务）列表。如果接收到的这个 GTID 列表与 master 上执行的 GTID 列表不一致，slave 读取事件可能比 master 滞后。如果关闭 GTID，这个值为空。

#### Executed\_Gtid\_Set

slave 执行的 GTID（事务）列表。如果这个列表与 Retrieved\_Gtid\_Set 不一致，说明 slave 可能没有执行全部事务，或者有些事务是由 slave 发出的。如果关闭 GTID，这个值为空。

如果复制拓扑使用 GTID，在确定存在丢失事务或重大 slave 滞后问题的时候，

Retrieved\_Gtid\_Set 和 Executed\_Gtid\_Set 就非常重要。参见在线 MySQL 参考手册中“使用全局事务标识符复制”一节获取更多关于 GTID 的信息以及如何在拓扑中的服务器之间管理事务。

如果 slave 启用了二进制日志，使用 SHOW BINARY LOGS 命令查看 slave 上可用的 binlog 文件列表及其大小（单位为字节）。示例 13-6 显示了 SHOW BINARY LOGS 命令的典型结果。



使用 FLUSH LOGS 命令可以轮换 slave 上的中继日志。

示例13-6: slave上的SHOW BINARY LOGS命令

491

```
mysql> SHOW BINARY LOGS;
+-----+-----+
| Log_name          | File_size |
+-----+-----+
| slave-bin.000001  | 5151604   |
| slave-bin.000002  | 1030108   |
| slave-bin.000003  | 1030044   |
+-----+-----+
3 rows in set (0.00 sec)
```

如果 slave 启用了二进制日志而且指定了 log\_slave\_updates 选项，还可以使用 SHOW BINLOG EVENTS 命令显示 slave 上的二进制日志事件。示例 13-7 显示了一个典型复制配置中的 binlog 事件。

在 MySQL 5.5 及其以后的版本中，还可以使用 SHOW RELAYLOG EVENTS 命令检查 slave 的中继日志。

示例13-7: SHOW BINLOG EVENTS 命令（基于语句的复制）

```
mysql> SHOW BINLOG EVENTS IN 'slave-bin.000001' FROM 2701 LIMIT 2 \G
***** 1. row *****
    Log_name: slave-bin.000001
      Pos: 2701
Event_type: Query
Server_id: 1
End_log_pos: 3098
    Info: use `employees`; CREATE TABLE salaries (
  emp_no      INT          NOT NULL,
  salary      INT          NOT NULL,
  from_date   DATE         NOT NULL,
  to_date     DATE         NOT NULL,
```

```

KEY          (emp_no),
FOREIGN KEY (emp_no) REFERENCES employees (emp_no) ON DELETE CASCADE,
PRIMARY KEY (emp_no, from_date)
)
***** 2. row *****
Log_name: slave-bin.000001
Pos: 3098
Event_type: Query
Server_id: 1
End_log_pos: 3405
Info: use `employees`; INSERT INTO `departments` VALUES
('d001','Marketing'),('d002','Finance'),
('d003','Human Resources'),('d004','Production'),
('d005','Development'),('d006','Quality Management'),
('d007','Sales'),('d008','Research'),
('d009','Customer Service')
2 rows in set (0.01 sec)

```

492

## slave 的状态变量

监控 slave 只有几个状态变量,包括在 master 上执行与 slave 相关的命令的次数的计数器,以及重要 slave 操作的统计数据。这里列出的前 4 个变量是与 slave 相关的命令的计数器。这些值应该与 slave 的维护频率相对应。如果它们不一致,可能需要检查以下两种情况:拓扑结构上 slave 数量是否比预期的多,或者某个 slave 重启次数是否过于频繁。这些变量包括:

Com\_show\_slave\_hosts

SHOW SLAVE HOSTS 命令执行的次数。

Com\_show\_slave\_status

SHOW SLAVE STATUS 命令执行的次数。

Com\_slave\_start

SLAVE START 命令执行的次数。

Com\_slave\_stop

SLAVE STOP 命令执行的次数。

Slave\_heartbeat\_period

master 的心跳检测的间隔时间的当前配置信息。

Slave\_last\_heartbeat

最近收到的心跳事件。显示为一个时间戳值。如果当前值小于当前值加上 Slave\_



heartbeat\_period 的和, 心跳事件可能会延迟。如果滞后很多说明 master 连接可能有问题。

### Slave\_open\_temp\_tables

Slave 的 SQL 线程使用的临时表的数量。该值如果过高, 说明 slave 负载过重。

### Slave\_received\_heartbeats

从 master 得到回复的心跳数, 该值应该与心跳检测间隔时间一致, 因为 slave 的重启会在心跳间隔时被断开。

### Slave\_retried\_transactions

Slave 启动后 SQL 线程重试事务的次数。

### Slave\_running

如果 slave 连接到 master, 而且 I/O 和 SQL 线程无错误地执行, 则该值为 ON, 否则为 OFF。

## 使用 MySQL 工作台监控复制

493

你已经知道了如何使用 MySQL 工作台监控网络流量和存储引擎。MySQL 工作台还可以通过简单的列表导航监控状态变量和系统变量。图 13-1 显示了 MySQL 工作台系统管理状态和系统变量页面, 其中高亮显示了复制状态变量。通过这个页面可以快速查看复制的状态。

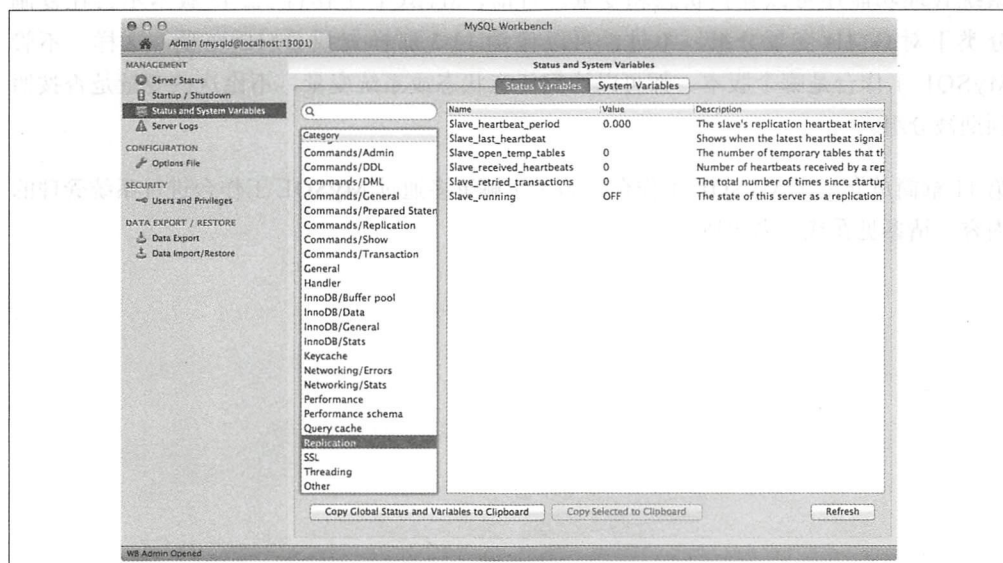


图13-1: 复制状态变量

MySQL 工作台还可以查看复制系统变量及它们的值。如果单击“系统变量”选项卡，就会在左边看到一长串分类列表。选择任意复制分类，可显示相关的系统变量。图 13-2 给出了 slave 复制的系统变量。注意某些变量有 [rw] 前缀，这表示变量是可读写的，因此可以在运行时修改它们。

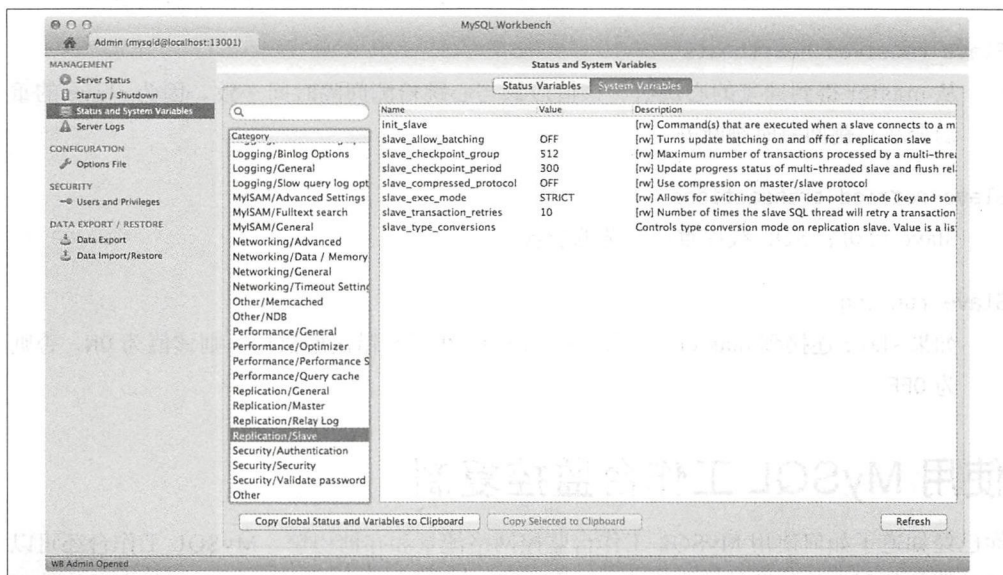


图 13-2: 复制系统变量

494 系统管理功能还可以查找状态和变量。目前，MySQL 工作台 5.2.45 版本不会在复制分类下对 GTID 变量分组。不过，可以像图 13-3 那样查找 GTID 变量。这样，不管 MySQL 工作台是哪个版本，都可以访问任意状态或系统变量，不论这些变量是否按照预期被分组。

第 11 章简单介绍了 MySQL 工作台。为了了解更多通过 MySQL 工作台进行系统管理的内容，请参见在线参考手册。

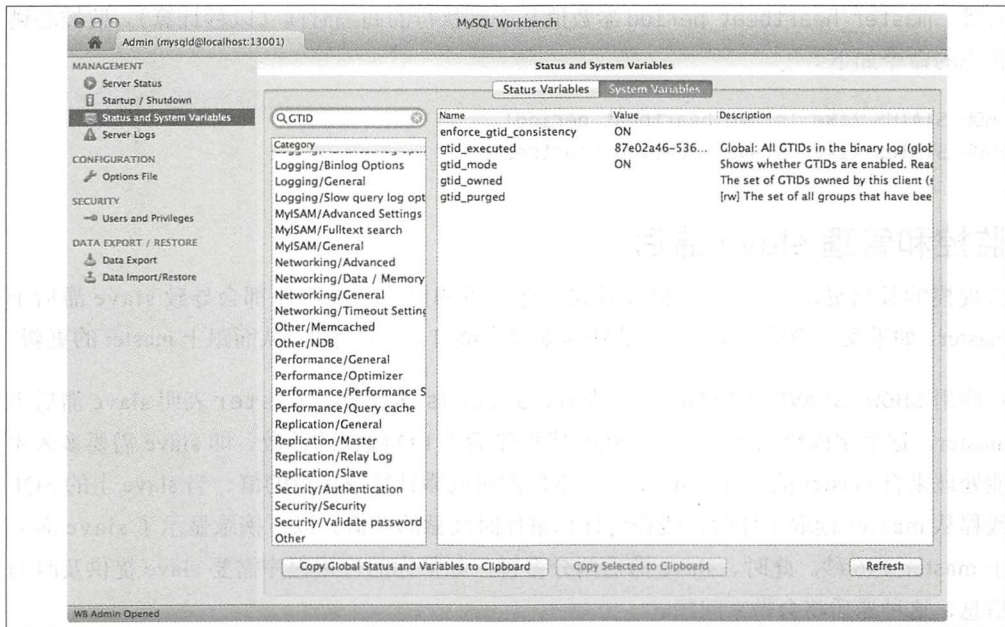


图13-3: GTID系统变量

## 其他需要考虑的问题

495

本节讨论监控复制需要考虑的其他问题，包括特定的网络因素和监控滞后（复制延时）。

### 网络

如果网络带宽有限、带宽竞争激烈或者连接缓慢，可以使用压缩提高复制性能。使用 `slave_compressed_protocol` 变量可配置压缩信息。

如果网络带宽不是问题，而且需要加密 master 到 slave 的数据，这时可以使用 SSL 连接。使用 `CHANGE MASTER` 命令可配置 SSL 连接。参看在线 MySQL 参考文档中的“使用 SSL 设置复制”一节，可了解更多关于复制中使用 SSL 连接的信息。

还有一个需要考虑的网络配置，即 master 心跳的使用。前面我们在 `SHOW SLAVE STATUS` 命令中已经看到了这个信息。心跳机制自动检测 master 与 slave 之间的连接状态，可以检测到毫秒级别的连接。复制场景中的 master 心跳机制，slave 必须与 master 同步，并且（几乎）没有延迟。心跳机制可以在 slave 上的复制停止之前，确保延迟在一定时间内被诊断出来。

496

使用 `CHANGE MASTER` 命令中的参数和 `master_heartbeat_period=<value>` 配置 master 的



心跳，`master_heartbeat_period` 参数值是心跳执行的间隔时间（以秒计算）。监控心跳状态的命令如下：

```
SHOW STATUS like 'slave_heartbeat period'
SHOW STATUS like 'slave_received_heartbeats'
```

## 监控和管理 slave 滞后

大规模的数据更新、slave 负担过重或其他严重的网络性能事件都会导致 slave 滞后于 master。如果发生滞后，slave 将无法快速处理中继日志中的事件，从而跟上 master 的更新。

可使用 `SHOW SLAVE STATUS` 命令查看，`Seconds_Behind_Master` 表明 slave 滞后于 master。这个字段给出了 slave 的 SQL 线程滞后于 I/O 线程的秒数，即 slave 需要多久才能处理来自 master 的事件。slave 使用事件的时间戳计算该字段的值。当 slave 上的 SQL 线程从 master 读取事件时，线程将计算事件时间戳的差值。以下摘录显示了 slave 滞后于 master 146 秒。此时，slave 滞后两分多钟。如果你的应用程序需要 slave 提供及时的信息，这种滞后将会带来问题。

```
mysql> SHOW SLAVE STATUS \G
...
      Seconds_Behind_Master: 146
...
```

`SHOW PROCESSLIST` 命令（在 slave 上运行）也表明了 slave 延迟的时间。这里我们看到 SQL 线程滞后的时间秒数，由最近复制事件的时间戳与 slave 运行该事件的实际时间之间的差计算而得。例如，如果 slave 在失去连接 30 分钟后再重新连接 master，则 `SHOW PROCESSLIST` 的显示结果中的 `Time` 字段应为 1800 秒左右，如下面的片段所示。这里 `Time` 值越大说明 slave 延迟越严重，将会产生过时数据。

```
mysql> SHOW PROCESSLIST \G
...
      Time: 1814
...
```

**497** 根据复制拓扑的设计方式不同，你可能需要复制数据以达到负载均衡。在这种情况下，一般使用多个 slave，将一部分应用程序或用户的 `SELECT` 查询分配给 slave 执行，从而减少 master 的负担。

## slave 滞后的原因和预防措施

slave 滞后令一些复制用户感到厌烦。slave 滞后的主要原因是 slave 是单线程的（事实



上有两个线程，但是只有一个线程执行事件，这是 slave 滞后的主要原因)。例如，多核 master 可以并行运行多个事务，比在单线程运行事务（二进制日志中的事件）的 slave 快。我们已经讨论过一些检测 slave 滞后的方法，本节将讨论一些 slave 滞后的常见原因及减少滞后的解决办法。



使用多线程 slave 可以缓解这些问题。

slave 滞后的原因有几个（例如网络延时）。可能是 slave 的 I/O 线程读取日志中的事件被延迟。最常见的滞后原因是 slave 在单线程中执行所有事务，而 master 却有很多线程并行执行事务。其他的原因还包括带有低效率 JOIN 的长查询、磁盘读取的 I/O 限制、锁竞争和 InnoDB 线程并发问题等。

现在我们知道了导致 slave 滞后的原因，下面让我们学习如何最小化 slave 滞后。

#### 组织数据

通过数据规范化和使用数据分片实现分布式，可以提高性能。这有助于减少数据的副本，但是就像在第 11 章中提过的，实际上有些数据的副本（如查找文本）可以提高系统性能。大致思想是恰到好处地使用足够的规范化和数据分片来提高系统性能，而不能过度。这只有靠数据拥有者自己通过经验或试验来确定。

#### 分而治之

我们知道，额外增加 slave 去处理查询（横向扩展）是提高性能的好办法，但是如果 slave 执行的查询非常多，横向扩展得不充分仍会导致 slave 滞后。最坏的情况是所有的 slave 都出现滞后。为解决这个问题，考虑使用复制过滤器分割数据，在不同的 slave 上复制不同的数据库。你仍可以使用横向扩展，但是在这种情况下，先使用中间 slave 处理过滤的数据库组，然后再进行扩展。

498

#### 识别并重构长时间运行的查询

如果长时间运行的查询导致 slave 滞后，考虑重构查询、操作或应用，发出较短的查询或更紧凑的事务。然而，如果你将该技术与复制过滤结合，就必须在发出跨复制过滤组的事务时留心。一旦某个本来应该是原子操作（一个事务）的长查询被分割，就可能带来数据完整性问题。

#### 负载均衡

还可以使用负载均衡将查询重定向到不同的 slave 中执行。这样可以减少每个 slave 执行查询的时间，从而有更多的计算时间去处理复制事件。

确保使用最新的硬件

显然，拥有最好的硬件常常为工作带来更好的性能。最起码，应该确保 slave 的硬件配置是最优的，至少要与 master 一样强大。

减少锁竞争

MyISAM 的表锁和 InnoDB 的行级锁可能导致 slave 滞后。如果查询导致大量 MyISAM 或者 InnoDB 表被锁定，请考虑重构查询，尽量避免使用锁。

## 使用 GTID

如果使用 GTID，在 master 和 slave 上可以使用 GTID 系统变量，确定事务已经在 master 上执行、被 slave 接收、然后在 slave 上执行。我们介绍一些与 GTID 相关的系统变量。下面列出了这些系统变量及其用法，注意如何在启用 GTID 的服务器上进行监控和故障排除的时候使用它们。

`enforce_gtid_consistency`

如果启动了这个变量，服务器禁止执行任何不安全的事务，包括在事务内部使用 `CREATE TABLE ... SELECT` 和 `CREATE TEMPORARY TABLE`。这个变量不是动态变量，默认情况下是禁用的，是只读的，是全局变量。

`gtid_executed`

这个变量是会话或全局范围的。如果在会话范围内使用，显示当前会话中写入缓存的一组事务。如果在全局范围使用，显示二进制日志中记录的全部事务。

499



发出 `RESET MASTER` 命令会在全局范围内清除这个变量。如果在 slave 上从 master 复制和恢复数据，可能需要执行重置命令。

`gtid_mode`

显示是否正在使用 GTID（值为 `ON`）。注意，随着未来 GTID 新功能的引入，服务器的这个变量可能有其他值加入。

`gtid_next`

确定 GTID 的创建方式。值为 `AUTOMATIC` 表示通过标准全局唯一机制创建 GTID。值为 `ANONYMOUS` 表示使用文件和位置生成 GTID，因此 GTID 不是唯一的。

`gtid_owned`

这个变量也有不同的范围。如果在会话范围使用，表示当前服务器拥有的所有 GTID

列表。如果在全局范围使用，表示所有 GTID 列表以及每个 GTID 的拥有者。

## gtid\_purged

显示已经从二进制日志中清除的事务。使用这个变量可以阻止 slave 执行某些事务。例如，如果从 master 复制或恢复数据，slave 接收然后执行 GTID，你可能不希望那些已经应用过的事务在 slave 上执行。所以，设置这个变量排除那些已经应用过的 GTID。

为某个指定数据库隔离特定的 GTID 是不可能的。因此，如果想要在 master 上做备份，然后在 slave 上恢复，需要 master 读取 GTID 集合，而且是整个 GTID 集合。在使用变量应用到 slave 的时候，是在告诉 slave GTID 列表中所有事务都要执行。因此，如果要部分恢复数据，必须十分小心。最好的办法是，在 master 上做全备份，设置 gtid\_purged 中的 GTID 列表，然后在 slave 上恢复。

## 小结

本章总结了很多监控 MySQL 的方法，提供了监控 MySQL 服务器各个方面的实战基础。

现在我们了解了操作系统监控、数据库性能及 MySQL 监控和基准的基础知识，拥有了可以成功优化服务器性能的工具和知识。

500

写复制问题报告的时候 Joel 笑了。他停下来看看门口，似乎觉得老板就要来了。

“Joel。”

Joel 吓了一跳，简直无法相信自己的预感。“先生，我已经解决了复制问题。”他赶紧说。

“很好！那你有空的时候把详细资料发给我吧。”

“我还发现订单处理系统有些很有趣的事情。”他注意到 Summerson 先生的眉毛微微扬起。Joel 继续说道：“看起来我们设置的缓冲池大小不正确。我认为我可以做些改进。”

Summerson 先生说：“又要监控？”

“是的，先生。我已经有了 InnoDB 存储引擎的报告，我把这个也发给你吧。”

“做得好，确实很好。”

Joel 了解老板的这种表情。老板想了两次，说明还有更多的工作。令 Joel 惊讶的是，老板只是慢慢地走开了。Joel 心想：“哦，看来我终于难倒他了。”

# 复制的故障排除

邮件的标题很简单——“修复西雅图的服务器”。Joel 知道如此神秘的邮件只可能是 Summerson 先生发来的。他快速看完邮件头，果然是 Summerson 先生，然后打开邮件开始阅读里面的内容。

“西雅图的服务器又出问题了，我看问题出在复制上。要把这件事当作首要任务来解决。”

“好，”Joel 喃喃自语。由于他上周生成的监控报告没有显示异常，所以他确定上次检查时复制的设置是正确的。Joel 不知道这个问题该如何下手，但是他知道去哪儿找答案。“看来我还是需要看看‘复制的故障排除’这一章。”

这时，一个熟悉的脑袋出现在门口。Joel 决定先发制人，“我已经在处理了。”听到这句话后，老板点点头，继续往走廊走去。

如果拓扑正在运行且正确配置的话，MySQL 的复制一般不会出什么问题，而且也不需要做什么调整。但是，也有出错的时候。有时错误很明显，就有明确的证据展开调查。有时候情况和问题本身也很容易理解，但是某些复杂问题的原因却不是那么明显。幸运的是，如果你遵循一些简单的排除复制故障的准则和做法，就可以解决这些问题。

本章重点介绍解决复制问题的技术。首先了解导致复制问题的原因，然后讨论能够帮助排除故障的基本工具，最后总结一些预防和解决复制问题的策略。



MySQL 集群复制的故障排除与本章介绍的流程一致。如果 MySQL 集群出了问题，请参看 9 章中的集群错误和启动问题的故障排除。

经验丰富的计算机用户知道计算系统容易出现偶然性错误。IT 专家认为应该防止错误，



确保为用户提供可靠的访问和数据，并将此作为他们的信条。然而，再妥善管理的系统也有可能出现问题。

MySQL 复制也不例外，特别是当 slave 状态不是崩溃安全 (crash-safe) 的时候。也就是说，如果 slave 上的 MySQL 实例崩溃，slave 就可能停止在一个未定义的状态。最坏的情况是，中继日志或 *master.info* 文件是损坏的。

事实上，拓扑越复杂（包括负载和数据库的复杂程度）、拓扑节点上角色越多样化，系统越容易出现错误。这并不意味着复制不能够扩展，相反，复制很容易被扩展成大规模的复制拓扑结构，这点我们已经讨论过。我们要说的是，当复制发生问题时，经常是由于意外操作或配置的改变引起的。

## 哪里出错了

很多事情出错都会导致复制故障。MySQL 复制最容易出现数据问题，比如数据损坏或者复制流意外中断。另外，系统崩溃导致 MySQL 不安全、不可控停机，也会导致复制重新启动的问题。

在问题修复做任何更改之前，应该总要对数据进行备份。在某些情况下，备份里包含已损坏或丢失的数据，但还是有好处的，特别是，无论你做什么，至少可以将数据恢复到发生错误当时的状态。否则你会惊奇地发现，将糟糕的事情变得更糟糕是一件多么容易的事情。

接下来的几个小节我们将通过描述 MySQL 复制的常见错误来讨论复制的故障排除。这些都是经常遇到的复制问题。虽然我们没有列出所有的复制问题，但是能够让你知道可能出错的类型。我们也简短描述了每个问题出现的可能原因。

## master 上的问题

◀ 503

虽然大多数错误出现在 slave 上，不过这一节我们介绍 master 上出现问题时的解决方案。有时候管理员会自然地怀疑是 slave 出错。在诊断复制问题时，应该同时检查 master 和 slave。

### master 崩溃及 Memory 表被占用

当 master 重新启动时，内存表中的所有数据都会被清除，这对于内存存储引擎而言是正常的。但是，如果内存存储引擎的表正在被复制，而 slave 还没有重启，slave 就会含有过时数据。

幸运的是，重启之后第一次访问内存表时，一个特殊的删除事件被发送到 slave 上，让 slave 清除过时数据，从而达到数据同步。但是，在表被引用和复制事件被传送之间的时间间隔内，还是会导致 slave 含有过时数据。为了避免这个问题，首先使用脚本清除数据，然后在启动时使用 `init_file` 选项向 master 重新填充数据。

例如，如果有一个内存表存储了频繁使用的数据，按照下面的例子创建文件，然后使用 `init_file` 选项引用它：

```
# Force slaves to purge data
DELETE FROM db1.mem_zip;
# Repopulate the data
INSERT INTO ...
```

第一个命令是删除查询，该命令在重启复制时将被复制到 slave 上。然后第二个语句是重新填充数据的语句。通过这种方法，可以确保即使 slave 在内存表中含有过时数据，也不会导致 master 和 slave 不一致。

## master 崩溃及二进制日志事件丢失

master 可能发生错误，而没有将最近发生的事件写入到磁盘上的二进制日志中。也就是说，如果服务器在 MySQL 把二进制事件缓存刷新到磁盘（二进制日志）之前而崩溃，这些缓存事件将会丢失。

通常这表明 slave 出错，说明二进制日志偏移事件丢失或不存在。在这种情况下，slave 将在重启后使用最近已知的二进制日志文件和 master 位置尝试重新连接 master，尽管二进制日志文件可能存在，但这些偏移不存在，因为增加偏移的事件没有被写入磁盘。

不幸的是，没有办法找回丢失的二进制日志事件。为了解决这个问题，必须检查 master 上当前 binlog 位置，然后告诉 slave 从 master 的下一个已知事件开始执行。为了确保 slave 是同步的，请一定要同时检查 master 和 slave 上的数据。注意下一个事件可能是下一个二进制日志文件中的第一个事件，所以要确定日志是否跨了多个文件。

master 上丢失的某些事件可能在系统崩溃之前已经被应用到数据上。应该对比 master 上有问题的表，确定 master 与 slave 之间是否有差别。这种情况很少见，但是如果某个行更新在 master 上执行了但却丢失了事件，稍后可能会产生问题，在 slave 上运行这个行更新将会导致错误。这时，slave 将试图更新一个不存在的行。

例如，考虑一个虚构的场景，有一个汽车经销商的简单数据库，该数据库包含新车和二手车销售信息的表，这些表中含有 `autoincrement` 键。

在 master 上执行：

```
INSERT INTO auto.used_cars VALUES (2004, 'Porsche', 'Cayman', 23100, 'blue');
```

在执行下面的语句之后发生了系统崩溃，但它还没有被写入二进制日志：

```
UPDATE auto.used_cars SET color = 'white' WHERE id = 17;
```

这时，更新查询在 master 崩溃时丢失。如果 slave 尝试重启，就会产生错误。使用以下建议解决这个问题。查看表的行数在 master 和 slave 上是否相同。注意，更新操作将 2004 保时捷的颜色从蓝色改为白色。现在，考虑当营业员试图帮助客户在 slave 上查找蓝色保时捷时，会发生什么事情：

```
SELECT * FROM auto.used_cars  
WHERE make = 'Porsche' AND model = 'Cayman' AND color = 'blue';
```

营业员会找到蓝色的保时捷吗？一个出色的汽车销售员总是能够通过肉眼观察确定现场是否有这辆车，但是我们假想一下，如果他太忙而没有时间去观察现场，而直接告诉顾客有他要的蓝色保时捷。可以想象，当顾客在试驾时发现车是白色时，该有多尴尬（还会失去一笔生意）。



为了防止数据在 master 崩溃时丢失，在系统启动时或在配置文件中启用 `sync_binlog`（将其设置为 1）。这个选项告诉 master 需要立即把事件刷新到二进制日志。虽然这可能导致 InnoDB 性能明显下降，但是如果你承担不起丢失任何数据（但可能丢失最新事件，这依赖于崩溃什么时候发生），这种保护措施是值得的。

虽然这个教学例子可能看起来并不严重，那我们考虑如果医学数据库或科学数据库丢失更新事件的情况。显然，丢失更新，甚至看上去很简单的更新，都会导致错误。事实上，这种场景就是数据损坏的一种形式。遇到这种问题时，要检查数据表的内容。如果 `sync_binlog=1`，崩溃恢复可以保证二进制日志和 InnoDB 是一致的，但是其对 MyISAM 表不起作用。

◀ 505

## master 上查询正常但在 slave 上出错

有时查询（例如更新或插入命令）可能在 master 上运行良好，而在 slave 上不能正常运行，从严格意义上讲，这并不是 master 的问题。导致这种错误的原因有很多，但大多数情况是参照完整性问题，或者 slave 或数据库的配置出了问题。

这种错误最常见的原因是 master 上更新的行在 slave 上不存在。此外，如果查询引用的

表不在 slave 上或者结构不同（不同的字段或不同的类型），也会出现这种错误。这时，必须更改 slave 使它与 master 一致，才能正确地执行查询。如果想要使用不同结构的表，一定要小心设计表，使数据只存在于 master，而 slave 上的查询无法访问（读取）。

在有些情况下，查询语句引用了一个没有被复制的表。例如，如果使用了任何复制过滤启动选项（快速检查 master 和 slave 的状态确认），那么查询语句引用的数据库可能在 slave 上不存在。在这种情况下，必须调整相应的过滤器，或者在 slave 上手动将丢失的表添加到丢失的数据库中。

其他情况下，查询失败的原因可能更复杂，如字符集问题、表损坏甚至数据损坏。如果你确定 master 和 slave 的配置一致，可能需要手动诊断查询。如果不能在 slave 上纠正问题，可能需要手动执行更新操作，让 slave 跳过包含失败查询的事件。



使用 `sql_slave_skip_counter` 变量在 slave 上跳过事件，并指定想要跳过的来自于 master 事件的数目。有时这是重新启动复制的最快办法。

## 崩溃之后表损坏

如果 master 或 slave 崩溃了，重启它们之后，你会发现有一个或多个表被损坏，或者被 MyISAM 标记为已崩溃，需要在重启复制之前解决这些问题。

506

通过检查服务器日志文件检查哪些表被损坏，像这样查找错误：

```
... [ERROR] /usr/bin/mysqld: Table 'db1.t1' is marked  
as crashed and should be repaired ...
```

使用以下命令一步完成优化和修复给定数据库（本例是 `db1`）中的所有表：

```
mysqlcheck -u <user> -p --check --optimize --auto-repair db1
```



MyISAM 表可以使用 `myisam-recover` 选项启动自动恢复。有 4 种恢复模式，详情参见在线 MySQL 参考文档。

一旦修复了受影响的表，还必须确定 slave 上的表是否被损坏。如果 master 和 slave 共享同一个数据中心，而且错误是由环境导致的（例如，它们被连接到同一个电源），这样做是必需的。





在修复前总是需要执行表的备份。在某些情况下，修复操作会导致数据丢失，或者使表处于未知状态。

此外，修复还会导致 master 与 slave 不同步，尤其当修复导致数据丢失的时候。可能需要对比受影响的数据表中的数据，确定 master 与 slave 是否同步。如果它们不同步，当 slave 丢失数据时，你可能需要为 slave 上受影响的表重新加载数据，或者当 master 丢失数据时，将 slave 上的数据复制到 master。

## master 上的二进制日志损坏

如果服务器崩溃或出现磁盘问题，导致 master 上的二进制日志被损坏，你将无法重启复制。二进制日志损坏的原因和类型有很多，任何损坏都会导致 slave 上的一个或更多事件无法执行，还会常常出现“不能解析中继日志事件”的错误。

在这种情况下，必须检查二进制日志中的可恢复事件，并使用 `FLUSH LOGS` 命令轮换 master 上的日志。结果，slave 可能会丢失数据，而且大多数情况下必然会出现错误。最好的恢复办法是使用可靠的备份和恢复工具重新同步 master 和 slave。除了轮换日志以外，507 要确保最小化数据丢失，使复制重启时不出现错误。

在某些情况下，如果很容易知道有多少事件损坏或丢失，可以使用 slave 上的 `sql_slave_skip_counter` 选项跳过这些损坏事件。通过对比 slave 上的 master 的 binlog 引用和 master 上的当前 binlog 位置来确认丢失或损坏的事件。方法之一是使用 `mysqlbinlog` 工具读取 master 的二进制日志，并统计事件。

## 杀死非事务型表上长时间运行的查询

如果强行终止修改非事务型表的查询，这个查询可能已经被复制到 slave 并被执行了。如果发生这种事情，master 上的更新可能与 slave 不同。

例如，如果你终止以下查询：更新一个含有 600 行数据的表中的 400 行，master 上只有 200 行被更新，而这时 slave 可能已完成了全部 400 行数据的更新。

因此，无论什么时候终止 master 上的数据更新查询，都需要确保这个查询没有在 slave 上执行。如果已经在 slave 上执行了（哪怕是作为预防措施），一旦更正了 master 上的表，就应该重新同步 slave 上的数据。通常在这种情况下需要修正 master，然后备份 master 上的数据，并在 slave 上进行恢复。

## 不安全的语句

如果数据库同时使用事务型表和非事务型表，只要对数据库或表的变更不是在同一事务中发生的，就不会遇到任何问题。但是，合并事务型表和非事务型表可能出问题，特别是使用不安全的语句的时候。尽管有些不安全的语句可以无错误地执行，但最好还是避免使用它们。

在线参考手册列出了很多不安全的语句。从根本上说，它们是能够在 master 和 slave 上导致不同结果的任何语句或从句。不安全的语句和函数有 `UUID()`、`FOUND_ROWS()`、`INSERT DELAYED` 和 `LIMIT`。例如，如果使用 `LIMIT` 的时候不使用 `ORDER BY` 从句，就会产生不同的结果。如果使用 `ORDER BY` 从句，并且表是一样的，结果就是可以预测的。

要了解全部的不安全语句，参见在线参考手册中“确定二进制日志中的安全和不安全语句”一节。

508 ▶ 尽管通过严密监控警告可以检测什么时候使用了不安全的语句，但是天真的用户还是会使用这些语句。例如，考虑下面的表：

```
CREATE TABLE test.t1 (a int, b char(20)) ENGINE=InnoDB;
```

现在假定几个不同的客户端同时向 master 写入以下更新，其中 master 上启用了基于语句的二进制日志。各个客户端由不同的 *mysql* 提示符区分。假定下面语句执行的顺序就是语句实际的执行顺序：

```
client_1> BEGIN;
client_1> INSERT INTO test.t1
VALUES (1, 'client_1'), (2, 'client_1'), (3, 'client_1');
client_2> BEGIN;
client_2> INSERT INTO test.t1
VALUES (4, 'client_2'), (5, 'client_2'), (6, 'client_2');
client_2> COMMIT;
client_1> COMMIT;
```

注意，虽然客户端 1 首先开始了事务，但是客户端 2 首先完成（提交）了事务。因为 InnoDB 保留了基于事务开始时间的插入顺序，所以结果表就是这样的（这里行的顺序与它们被插入 InnoDB 缓存机制的顺序一样，这就是 master 上不使用索引的 `SELECT` 结果）：

```
mysql> SELECT * FROM test.t1;
+-----+-----+
| a     | b           |
+-----+-----+
| 1     | client_1    |
```

```

| 2 | client_1 |
| 3 | client_1 |
| 4 | client_2 |
| 5 | client_2 |
| 6 | client_2 |
+-----+-----+
6 rows in set (0.00 sec)

```

但是，由于实际上客户端 2 上的事务是首先执行的，二进制日志就按照这个顺序存储，因此 slave 也根据这个顺序读取。所以，在连接的 slave 上，同一个表的顺序可能与原来无索引的顺序略有不同（但是数据是一样的）：

```

mysql> SELECT * FROM test.t1;
+-----+-----+
| a | b |
+-----+-----+
| 4 | client_2 |
| 5 | client_2 |
| 6 | client_2 |
| 1 | client_1 |
| 2 | client_1 |
| 3 | client_1 |
+-----+-----+
6 rows in set (0.00 sec)

```

509

现在，我们考虑在 master 上执行这样的不安全语句：

```
UPDATE test.t1 SET b='this_client' LIMIT 3;
```

这里，我们只想更改前 3 行的 b 字段。master 上期望的结果是：

```

mysql> SELECT * FROM test.t1;
+-----+-----+
| a | b |
+-----+-----+
| 1 | this_client |
| 2 | this_client |
| 3 | this_client |
| 4 | client_2 |
| 5 | client_2 |
| 6 | client_2 |
+-----+-----+
6 rows in set (0.00 sec)

```

但是，复制语句后在 slave 上我们得到的结果完全不一样。你会发现，这种不安全的语

句确实会产生一些挺有趣的令人沮丧的数据不一致性：

```
mysql> SELECT * FROM test.t1;
```

```
+-----+-----+
| a      | b          |
+-----+-----+
| 4      | this_client |
| 5      | this_client |
| 6      | this_client |
| 1      | client_1    |
| 2      | client_1    |
| 3      | client_1    |
+-----+-----+
```

```
6 rows in set (0.00 sec)
```

显然，应该避免不安全的语句。最好的办法之一是培训用户哪些语句是不安全的，监视复制警告，并考虑全部使用事务型表和基于行的日志。

## slave 上的问题

你遇到的大部分问题是由于 slave 出现错误引起的。在某些情况下，如前面所述，有些问题来源于 master，但总是会以这种或那种方式出现在 slave 上。接下来的小节描述了 slave 上出现的一些常见问题。

### 在 slave 上开启二进制日志

确保 slave 更强大的一个办法是使用 `log-slave-updates` 选项开启二进制日志。这将导致 slave 记录中继日志中执行的事件，然后创建一个二进制日志，在中继日志被损坏时，可以使用这个二进制日志文件重放 slave 上的事件。

## slave 服务器崩溃及复制无法启动

当 slave 服务器崩溃时，一旦知道 slave 上最后执行的正确事件，就能很容易地重新建立到 master 的复制。通过 `SHOW SLAVE STATUS` 命令可查看结果。

但是，如果遇到账号访问错误，复制可能无法重启。这可能是认证问题导致的，例如 slave 被重建了，还没创建复制账户（即 `CHANGE MASTER` 命令使用的账号）或者密码不正确。还可能是因为 master 或 slave 上的表被损坏而无法重启。在这种情况下，可以在 slave 的 MySQL 服务器的控制台和日志上看到连接错误信息。

当发生这种情况时，总是需要检查 master 上复制用户的权限。确保用户被授予正确的权



限，权限在配置文件中或通过 `CHANGE MASTER` 命令定义。权限差不多应该是这样的：

```
GRANT REPLICATION SLAVE ON *.*  
TO 'rpl_user'@'192.168.1.%' IDENTIFIED BY 'password_here';
```

根据需要修改这个命令，是解决以上问题的一种手段。

## slave 连接超时及反复重新连接

如果你的拓扑结构上有很多 slave，而且有两个或两个以上的 slave 都没有设置 `server_id` 选项或者它们的 `server_id` 值相同，那就可能出现服务器 ID 冲突。当发生这种情况时，其中一个 slave 可能会频繁超时或丢失后重新连接序列。

这个问题很简单，只是因为 slave 没有唯一 ID，但却很难被诊断出来（或者我们应该说，它很容易被误诊为连接问题）。总是应该检查 master 和 slave 的错误日志以查看错误信息，错误本身可能有超时问题。

为了防止这类问题的发生，总是需要确保在配置文件中或在启动命令行中为所有服务器都设置了 `server_id`。 ◀ 511

## slave 上的查询结果与 master 上的不同

一个更难以觉察的问题是：在一台或多台 slave 上执行的查询结果与 master 上的不一致。有可能你从来就不会注意到这个问题。它可能像排序问题一样简单或无害，也可能严重到导致结果集丢失行或者多了额外的行。

这类问题发生的主要原因可能是执行了不安全的语句、写操作发给了 slave、混合使用事务型和非事务型存储引擎，以及 master 和 slave 上的查询选项集不同（例如 `SQL_MODE`）。

较为不常见的原因是 master 和 slave 的字符集不同。例如，master 上的默认字符集为一种类型，而 slave 上的字符集被配置为其他类型。只要这两个字符集是兼容的（即为字符存储相同数量的字节），也是没问题的。但是，如果用户开始抱怨多了或丢失了行，或者结果的排序不同，你应该仔细检查 master 和 slave 上行结果不同的那些数据。

还有一个原因是，master 和 slave 上的默认存储引擎不同（例如，master 上使用 MyISAM 存储引擎，而 slave 上使用 InnoDB 存储引擎）。在这种情况下，如果没有主键或索引保证正确的顺序（无 `ORDER BY` 从句），那么查询结果完全有可能以不同的顺序出现。

一个更加不太可能的原因是 master 和 slave 上的表定义不同。例如，slave 上没有某些开始字段或结束字段。解决方法是永远在 `SELECT` 语句中指定所有列。例如，如果 slave

上的列比 master 多, slave 上的 `SELECT *` 查询就会比 master 的执行结果多。如果 slave 上还有一些额外的开始字段, 那么情况就很复杂了。这时, `SELECT *` 查询的结果就与 master 不一致了。

同理, 在 master 上执行 `INSERT INTO` 查询的时候不指定列, 也会出现这个问题, 可能问题更严重。同样, 如果 slave 上有更多的开始字段, 那么数据也会不一致, 查询可能会失败。最好将查询写成 `INSERT INTO <table> (a,b,c)...`, 明确列出所有字段。

512



列的顺序很重要。master 和 slave 上结构不同的表必须拥有相同的字段子集, 这些字段可能出现在表的开始或结束位置。

使用这个功能可能会遇到很多潜在的错误。例如, 如果 master 上的字段比 slave 多, 用户会以为所有字段的数据都复制了, 但其实 slave 上只有部分字段。有时候你确实希望 slave 上的字段少一些, 查询的时候谨慎一些也是可以的。但是粗心的用户会不小心删掉某些列, 使复制继续进行。有时候, 如果 slave 上执行的 `SELECT` 查询引用了这些不存在的列, 查询就会失败, 从而导致上述问题。还有些时候, 可能只是因为应用程序数据丢失了。

一个常见的用户错误是, 在 slave 上执行了表或数据库的其他变更操作, 但在 master 上没有执行 (也就是说, 用户在 slave 上执行了一些非复制的数据操作更新了表结构, 但在 master 上没有执行同样的操作), 这样会导致 master 和 slave 上的查询结果不同。如果发生这种情况, 查询可能会返回错误的结果、错误的列、错误的顺序或额外数据, 或者也有可能由于引用了不存在的字段而导致查询失败。总是要事先做好预防措施, 检查与这些问题有关的表的布局, 这样可以确保 master 和 slave 上的表一致。如果它们不一致, 请重新同步 master 和 slave 上的表, 然后重新执行查询操作。

## 当尝试重启 SSL 时 slave 出错

与 SSL 连接有关的问题一般都是前面提到的常见的权限问题。在这种情况下, 授予权限的时候还必须包括 `REQUIRE SSL` 选项, 如下所示 (务必检查复制用户是否存在而且拥有正确的权限):

```
GRANT REPLICATION SLAVE ON *.*  
TO 'rpl_user'@'%' IDENTIFIED BY 'password_here' REQUIRE SSL;
```

当使用 SSL 连接时, 与重启复制有关的其他问题是: 丢失认证文件, 或配置文件中与 SSL 相关的选项 (如 `ssl-ca`、`ssl-cert` 和 `ssl-key`) 的值设置不正确, 或 `CHANGE`

MASTER 命令中与 SSL 相关的选项（如 MASTER\_SSL\_CA、MASTER\_SSL\_CAPATH、MASTER\_SSL\_CERT 和 MASTER\_SSL\_KEY）的值设置不正确。务必检查设置和路径，确保自从上一次启动复制以来这些设置没有发生任何变化。

## 内存表数据丢失

如果一个或多个数据库使用内存存储引擎，当 slave 重启（服务器重启，不是 slave 线程重启）时，这些表里的数据将会丢失。这是意料之中的，因为内存表中的数据会在系统重启时被清除。这时，表的配置仍然存在，表也可以被访问，但是里面的数据已经被清空了。

当 slave 服务器被重启时，定向到内存表的查询操作（如 UPDATE 操作）就会失败，或者查询结果不准确（如 SELECT 操作）。因此，错误可能不会立即出现，可能只是在查询结果中丢失行而已。

为了避免该类问题，你应该小心使用数据库中的内存表。不要在没有准备好服务器重启或崩溃时的数据恢复策略的情况下，在 master 上创建内存表，这些内存表能够通过复制的方式在 slave 上被更新。例如，在复制之前执行一个脚本，该脚本从 master 复制数据。如果这些数据被获得，使用脚本将数据重新填充到 slave。

其他需要考虑的事情还有：在复制过程中过滤表，或者对任何复制表没有使用内存存储引擎。

## slave 崩溃后临时表丢失

如果复制数据库和查询时使用了临时表，应该考虑一些有关临时表的重要因素。当 slave 重启时，临时表将丢失。如果从 master 复制了任何临时表，从那一刻起就不能重启 slave，需要手动创建表或跳过那些引用该临时表的查询操作。

这种情况经常导致查询在一个或多个 slave 上无法执行。解决办法与处理内存表丢失的办法类似。具体来说，为了使这些查询能够被执行，需要手动重建临时表，或者将 slave 上的数据与 master 上的数据重新同步，并在重启 slave 的时候跳过这些查询。

## slave 运行慢而且与 master 不同步

slave 滞后，又称过度滞后（excessive lag），是指 slave 处理来自 master 的所有事件的速度不够快，从而导致数据更新的延时。在极端情况下，slave 上的数据更新会过期而导致不正确的结果。例如，如果票务代理系统中的 slave 比 master 延迟好几分钟，这个票务系统可能会出售已经不存在的座位（例如，这些座位在 master 上已经被标记为“已售



出”，而 slave 由于延时而没有更新）。

我们在前面的章节中讨论过这个问题，不过这里仍然总结一下解决方案。要检测这个问题，需要监控 slave 的 `SHOW SLAVE STATUS` 输出结果，检查 `Seconds_Behind_Master` 字段，确保它的值在应用程序允许的范围之内。为了解决该问题，考虑将某些数据库移到其他 slave 上，从而减少复制到 slave 的数据库数量，减少网络延迟（如果有的话），并提高数据存储性能。

例如，为繁重或昂贵的数据变更添加额外的 slave，可以使 slave 免于处理一些不相关的事件。减轻复制负担的方法还有：在一个单独的 slave 上做变更，然后在拓扑结构中的所有其他机器上使用可靠的备份和恢复方法来应用这些变更。

## slave 崩溃后数据丢失

slave 可能会崩溃从而没有记录最近可知的 master 的 binlog 位置。这些信息存储在 `relay_log.info` 文件中。如果发生这种情况，slave 将试图在错误（旧）的位置重启，试图执行那些已经执行过的查询。这常常会造成查询错误，可以通过跳过重复事件来解决。

然而，这些重复事件也可能导致数据被改变（被损坏），致使 slave 与 master 不同步。不幸的是，这类问题不太容易被检测出来。仔细检查日志文件，可能会发现有些事件已经被执行过了，需要检查 binlog 事件和 master 的二进制日志以确定哪些事件是重复的。



MySQL 服务器有一个基于 Python 的工具集，叫 MySQL 实用工具（MySQL Utilities）。其中有一个工具用于识别数据和结构的不同。MySQL 实用工具将在第 17 章进行详细讨论。

## 崩溃后表损坏

在崩溃后重启 master 的时候，你可能会发现一个或多个表被损坏，或被 MyISAM 标记为崩溃。需要在重启复制之前解决这些问题。一旦修复了这些受影响的表，就保证了 slave 上的表不会因为修复而导致数据丢失。这种情况很少发生，但是还是应该检查一下。如果可疑，总是需要在重启复制之前使用备份和恢复或类似操作手动将这些表与 master 同步。



在硬件或服务器崩溃的过程中，当发生部分页面写操作时，MyISAM 上执行恢复操作之后很可能出现数据丢失。不幸的是，数据是否丢失不太容易被检测出来。



## slave 上中继日志损坏

如果服务器崩溃或出现磁盘问题，将会导致 slave 上的中继日志损坏，复制将会由于某个与中继日志相关的错误而停止。中继日志损坏的原因和类型很多，但是都会导致 slave 上的一个或多个事件无法执行。

当这种情况发生时，恢复的最佳选择是：确认 master 二进制日志最后执行的已知事件的位置，然后使用 `CHANGE MASTER` 命令重启复制，前提是已知 master 的 binlog 信息。这将强迫 slave 重新创建新的中继日志。不幸的是，这意味着旧中继日志的任何恢复都是不合规矩的。

## slave 重启时的多个错误

难以检测和修复的难题之一是：在 slave 初次启动或重新启动时产生多个错误。可能遇到的错误很多，有时候是随机发生的，或者没有明确可识别的原因。

当这种情况发生时，检查错误日志和 `SHOW SLAVE STATUS` 的输出结果，寻找 slave 连接和复制启动过程中报告的相关错误信息。还可以检查 master 和 slave 上的 `max_allowed_packet` 的大小。如果 master 上该参数的值比 slave 大，那么 master 上记录的事件的大小可能比 slave 大。这会导致看起来不合逻辑的随机错误。

## slave 上事务失败的后果

一般情况下，如果事务失败，更新被回滚以避免部分更新问题。然而，如果混合使用事务型和非事务型表，情况就很复杂——事务型更新被回滚，但是非事务型更新不回滚。这样将导致以下问题：数据丢失、数据重复、数据冗余或非事务型表的无用更新等。

避免该类问题的最好办法是避免在数据库中混合使用事务型表和非事务型表，并且总是使用事务型存储引擎。

## I/O 线程的问题

有三种常见的与 I/O 线程有关的问题：

- 到 master 的连接丢失
- 到 master 的连接断断续续
- slave 严重滞后于 master

从 slave 状态输出的 `Last_IO_Errno` 和 `Last_IO_Error` 字段可以看到 master 连接错误。而且，`Slave_IO_Running` 的状态值可能为 `No`。这些错误的原因从错误描述中应该可以明

显看出来。如果错误不那么容易理解，就要按照以下步骤隔离问题：

- 检查 `CHANGE MASTER` 命令，确保使用的是正确的用户证书。
- 检查 master 上复制用户的证书。
- 在某个客户端使用 `ping hostname` 检查网络，确保 master 是可达的，其中 `hostname` 是 master 的主机名。
- 使用 `telnet hostname n` 检查服务器上使用的端口是可达的，其中 `n` 是 master 的端口。

如果到 master 的连接丢失了，而 slave 能够重新建立连接，那么问题就与网络带宽有关了。这时，使用网络测试工具确保带宽足够，并采取适当的措施减少带宽的过度使用。

如果 slave 滞后 master 很多，应该确保中继日志没有损坏，slave 没有错误或警告，以及 slave 的运行正确（一个常见问题是系统被另一个进程拖累）。如果这些都检查了没问题，考虑使用多线程 slave 或复制过滤，在 slave 组之间隔离数据库。

## SQL 线程的问题：不一致

SQL 线程最常见的问题是语句在 slave 上执行失败。这时，slave 的 SQL 线程会停止，slave 状态中会呈现这个错误。检查 slave 状态寻找线索，搞清楚语句失败的原因，并采取纠正措施。

slave 上的语句错误（而 master 上没有）的原因前面已经讲过了。通常原因是键冲突，不小心向 slave 发送写请求，或者 slave 和 master 上的数据不一样。无论何时遇到这些问题，如果可以试着阻止以后发生这种事情的话，都应该跟踪问题的原因。重要的一步是确定向 slave 发出数据更新操作的应用程序或用户。

大部分这类问题的解决办法是将 slave 与 master 同步。检查 master 和 slave 之间的一致性有一个小巧的工具，是 MySQL 实用工具里的 `mysqldbcompare` 脚本。这是一个离线工具，能够在活动不频繁的时候在 slave 上运行，或者如果将表锁定一段时间是安全的情况下也可以运行这个工具。这个工具不仅可以发现那些结构不同或丢失的对象，还可以找出不一样的数据行，而且能够生成转换语句，用于将 slave 恢复到与 master 一致的状态。这个工具和其他一些复制工具将在第 17 章中详细讨论。

517

## slave 上的错误不一样

尽管不常见，但是可能遇到这种情况：语句在 slave 和 master 上呈现的错误不一样。错误通常是这样的：“查询 X 导致 master 和 slave 上的错误不同...”这个消息给出了错误，但可能只是简单的类似“错误 0”这样的消息，这表明语句在 slave 上成功了但在 master

上有错误。

出现这个问题最可能的原因是 master 上的触发器或事件发生错误。例如，如果 master 上的触发器失败了，但是另一个等效的触发器在 slave 上成功了，或者 slave 上的触发器不见了（这个更常见）。这时，最好的办法是更正 master 上的错误，然后告诉 slave 跳过这个语句。

但是，问题也可能与触发器中的死锁有关，这是由数据不一致引起的。如果发生这种情况，需要首先更正不一致，然后重试或者跳过语句。

## 高级复制问题

更高级的复制拓扑结构中存在一些天然的复杂性。这一节我们讨论使用高级复制功能时可能遇到的常见问题。

### 变更没有在拓扑中复制

有时候，某个数据库对象的变更没有被复制。例如，`ALTER TABLE` 可能被复制了，但是 `FLUSH`、`REPAIR TABLE` 及类似的维护性命令没有被复制。只要发生这种情况，请查阅数据操控命令（DML）和维护命令的限制性。

导致该问题发生的典型原因是：无经验的管理员或开发者试图管理 master 上的数据库，并希望 master 上的更新被复制到 slave 上。

只要对数据库对象做了重大变更，如在文件级别或使用维护命令改变其结构，请在所有的 slave 上执行这些命令或步骤，以确保更新在整个拓扑结构中传播。

通常，专业的管理人员使用脚本完成这件事，作为日常维护工作。一般来说，脚本按顺序停止复制、应用更新，然后自动重启复制。

518

### 环形复制的问题

如果使用循环复制，而且已经从复制错误中恢复，其中该复制错误是由于一个或多个服务器从拓扑中离开，此时，会遇到这样的问题：某个事件在有些服务器上被执行不止一次。如果查询失败（如键冲突），将导致复制失败。原因是源服务器已经被移出拓扑。

如果发生这种情况，源服务器无法终止事件的复制。使用 `IGNORE_SERVER_IDS` 选项（MySQL 5.5.2 及其后的版本可用）和 `CHANGE MASTER` 命令可以解决该问题，指定一组忽略该事件的服务器 ID 列表。如果丢失的服务器被恢复，必须调整这个设置，不再忽略那些来自于替代服务器的事件。



## 多 master 的问题

使用循环复制（多 master 拓扑结构的特殊形式）时，如果你正在恢复复制错误，可能会发现有些事件被执行多次。这些事件通常来自于被移出的服务器。像解决循环复制问题那样解决这个问题——使用 `CHANGE MASTER` 命令将被移出的服务器的 ID 列表放到 `IGNORE_SERVER_IDS` 选项中。

多 master 复制还有一个问题：如果对同一个表的变更同时发生在两个 master 上，而且该表的主键为自增列时，这时，就会遇到重复键错误。如果必须在多个 master 上插入新行，请使用 `auto_increment_increment` 和 `auto_increment_offset` 选项进行主键值的递增。例如，某个服务器仅仅只能以偶数递增，而另一个服务器以奇数递增。如果不止两个 master 同时更新同一个带有自增主键的表，那么情况更加复杂。不仅使键值自增更加困难，而且如果需要替换拓扑结构中正在更新表的服务器，那就变成了一个管理性问题。如果增量太大（而且数据量非常大），某个大表的主键类型值可能会越界。

### 519 HA\_ERR\_KEY\_NOT\_FOUND 错误

该错误在基于行的复制拓扑中很常见。最有可能导致该错误发生的原因是：需要被更新或删除的行并不存在或已经被改变，以至于存储引擎找不到它。该错误主要发生在循环复制中，或者直接对 slave 上的复制数据进行变更时。当这种情况发生时，必须找出冲突的根源并修复数据或跳过有问题的事件。

## GTID 问题

全局唯一事务标识符（GTID）是复制的最新功能。前面我们已经讨论过，GTID 能够解决故障转移问题，应对拓扑中 master 的损失。不幸的是，复制中会遇到一些与 GTID 有关的问题，它们自身的属性使它们很稳定。但是，在启用了 GTID 的 slave 运行的过程中，还是可能会遇到问题。

最常见的情况是，你不会遇到错误，但是会发现 slave 的事务集合不完整。这时，使用前面讨论过的技术确定哪些 GTID 已经在 slave 上执行过。如果等不到 slave 赶上 master，则使用新的复制协议使另一个 slave 赶上来，让这个 slave 成为另一个最新的 slave。这样，slave 就会滞后于那些新 master 上丢失的 GTID 请求。一旦 slave 跟上了，就可以将它返回到原来的 master。

可能遇到的与 GTID 有关的错误的一种情况是，试图供应（provision）slave。如果 slave 比 master 滞后，或者并没有 master 上的全部数据，通常应该在 slave 上恢复 master 的一个最近备份。还可以这么做，不过需要在 slave 上设置 `gtid_purged` 变量，将 master 上



执行的全部 GTID（即 master 的 `gtid_executed` 系统变量值）都包含进来。

如果使用 `mysqldump` 或者 MySQL 实用工具中的 `mysqldbcopy`、`mysqldbexport` 和 `mysqldbimport` 命令，正确的 GTID 语句会自动生成。但是，如果想要在 slave 上设置 `gtid_purged` 变量，可能遇到类似“GTID\_PURGED 只能在 GTID\_EXECUTED 为空时设置”的错误。这时因为目标服务器并不是处在干净的复制状态。要更正这个错误，首先在 slave 上发出 `RESET MASTER` 命令清空 `gtid_purged` 变量，然后设置 `gtid_purged` 变量。



在使用 GTID 的主动复制拓扑中，千万不要在 master 上执行 `RESET MASTER` 命令，使用这个命令的时候务必非常小心。

◀ 520

## 复制的故障排除工具

如果你已经使用或建立了复制，或进行了维护工作，你会熟知很多诊断和修复复制问题的工具。

本节我们讨论诊断复制问题的工具，并提供一些如何及何时使用这些工具的建议。

### SHOW PROCESSLIST

如果遇到问题，总是应该查看哪些进程正在运行。这个命令告诉你与复制相关的每个进程的当前状态。在检查问题的时候首先检查这里。

### SHOW MASTER STATUS 和 SHOW SLAVE STATUS

这些 SQL 命令是诊断复制问题的主要工具。它们与 `SHOW PROCESSLIST` 命令一起使用，先在 master 上执行这些命令，然后在 slave 上执行，再检查输出结果。slave 命令有一个扩展参数集，在诊断复制问题时非常有用。

### SHOW GRANTS FOR <replication user>

只要遇到 slave 用户访问问题，就应该首先检查 slave 用户的权限，确保它们没有被更改过。

### CHANGE MASTER

有时候配置文件在知情或不知情的情况下被改变了，使用这个 SQL 命令重写最后已知的连接参数，更正 slave 连接问题。

### STOP/START SLAVE

使用这些 SQL 命令启动和停止复制。如果 slave 处于错误状态，有时候停止 slave

也是个好办法。

#### 检查配置文件

有时候问题的发生是由于未批准或遗忘的配置变更造成的。在诊断连接问题时，例行检查配置文件。

#### 521 检查服务器日志

应该在诊断问题时养成检查服务器日志的习惯。检查服务器日志有时可以发现一些其他地方看不到的错误。虽然有时候很隐晦，但是这些错误和警告信息是非常有用的。

#### SHOW SLAVE HOSTS

如果它们使用了 `report-host` 选项的话，使用该命令可识别 master 上连接的 slave。

#### SHOW BINLOG EVENTS

该 SQL 命令显示二进制日志中的事件。如果使用基于语句的复制，该命令将显示 SQL 语句的变更。

#### *mysqlbinlog*

该工具允许读取二进制日志或中继日志中的事件，通常说明什么时候出现了损坏事件。在诊断与事件和二进制日志相关的问题时，请毫不犹豫地使用这个工具。

#### PURGE BINARY LOGS

该 SQL 命令允许删除二进制日志中的某些事件，比如在特定时间或给定事件后发生的事件。例行维护计划应该包括使用该命令清除那些不再使用的旧二进制日志。应该指定明确的持续时间，防止不小心或过于频繁地清空二进制日志。

现在我们已经回顾了了在复制中可能遇到的一些问题，并了解了在典型 MySQL 版本中可用的诊断工具列表，接下来我们讨论解决复制问题的策略。

## 最佳实践

回顾复制中可能发生的潜在问题，以及列举的解决这些问题的工具，这些只是完整解决方案的一部分。还有一些经过验证的策略和最佳实践能够快速解决复制问题。

这一节描述这些策略和最佳实践，在诊断和修复复制问题时需要培养。我们并没有按特定的顺序来讲述——这取决于想要解决什么问题，其中一个或多个可能会对你有所帮助。

## 了解你的拓扑结构

如果你在少数服务器上使用 MySQL 复制，将拓扑配置存储在内存中并不是什么难事。

可能与单 master 和一个或多个 slave 的拓扑结构一样简单，或者也有可能像两个服务器的多 master 拓扑结构那样复杂。但是，要记住，拓扑结构和所有的配置参数，总有一天会变得不可能。

拓扑结构和配置越复杂，就越难确定问题发生的原因，以及从哪里开始修复操作。在一个拥有上百台 slave 的拓扑系统中很容易忽略单个 slave。

最好是拥有一份拓扑结构的图谱及其当前配置设置。应该在记事本或文件中记录一份复制启动的相关信息，并将该记录放在大家比较容易找到的地方。对于那些理解复制管理却不了解你的安装配置的人来说，这个信息非常有价值。

你应该拥有一份拓扑系统的文本或图形形式的图纸，在上面标明过滤器（master 和 slave），以及各个服务器在拓扑结构中的角色。还要包含 CHANGE MASTER 命令、所有的选项及所有服务器的配置文件内容。

拓扑图纸不需要很详细，也不需要是什么艺术奇作，简单的线条描画就可以了。图 14-1 显示了一个混合拓扑结构图，其中标注了过滤器和角色记号。

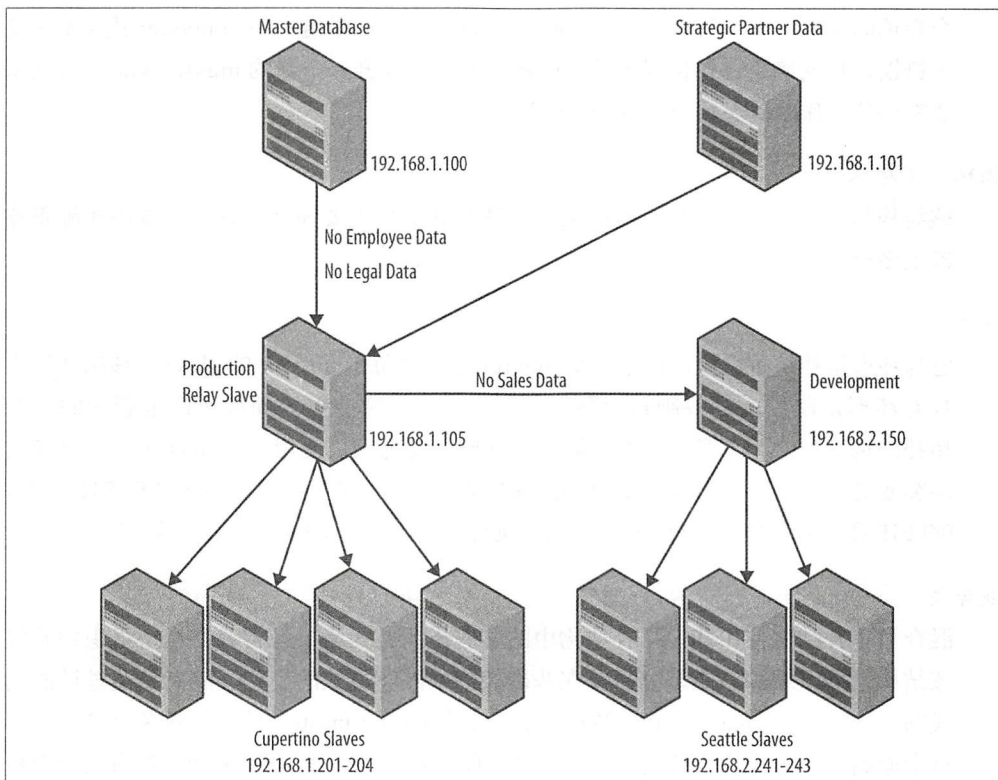


图14-1: 拓扑图例

注意, 中继 slave (192.168.1.105) 有两个 master (192.168.1.100 和 192.168.1.101)。这显得很奇怪, 因为没有一个是 slave 可以拥有一个以上的 master。为了达到这种级别的整合 (从第三方消费数据), 在中继 slave 上需要第二个 MySQL 服务器实例, 从战略伙伴 (192.168.1.101) 复制数据, 然后在中继 slave 上使用脚本定期将 MySQL 的第二个实例上的数据转移到 MySQL 的主实例上。这样将可以达到图 14-1 所示的集成方式, 加上一些手动工作还有战略伙伴数据的延迟性更新。

特定的拓扑系统会产生特定的问题, 这些问题我们在前面章节也讨论过很多次。下面我们列出不同类型的拓扑结构, 包括这些类型的简单描述以及需要注意的常见问题。

#### 星形 (又称单 master)

这是典型的单 master、多 slave 的拓扑结构。除了那些能够作用于整体复制拓扑的错误, 没有什么特殊的限制和问题。然而, 将 slave 提升为 master 比较复杂, 因为必须将最新的 slave 提升为 master。这很难决定, 可能需要检查每个 slave 的状态。

#### 链式

在这种结构中, 一个 slave 是另一个 slave 的 master, 以此类推, 最后的端点只是一个简单的 slave。除了前面提到的服务器 ID 问题, 如果链中间的 master 或 slave 发生错误, 如何确定错误位置也是个问题。此外, 提升一个新的 master/slave 节点需要额外的工作以确保整个系统的一致性。

#### 523 循环 (又称环形)

该结构与链形类似, 但是没有结束点。该拓扑结构需要谨慎配置, 使事件在源服务器上终止。

#### 多主

它与环形拓扑结构类似, 但是每个 master 是其他每个 master 的 slave。该拓扑结构具有环形复制的所有限制和问题, 而且会出现冲突变更因而成为最难管理的复杂拓扑结构之一。为了避免冲突问题, 必须确保变更仅仅使用一个 master。如果发生冲突变更, 比如在一个 master 上执行 UPDATE 命令, 在另一个上执行 DELETE 命令, DELETE 命令有可能在 UPDATE 命令之前执行, 这会阻止 UPDATE 命令更新任何行。

#### 混合型

混合型拓扑结构使用其他拓扑结构中的某些或所有元素。经常会在大型组织中看到该结构, 该组织的功能划分需要在基础架构中隔离类似的元素。隔离是通过过滤完成的, 数据从原始 master (有时被称为 prime 或 master master) 被划分到多个 slave 中, 这个原始 master 包含了发送到 slave 的所有数据, 这些 slave 又成为它们自己星形拓扑的 master。它是目前最复杂的拓扑结构, 需要持续更新文档。



## 检查所有服务器的状态

最好的预防性措施之一是定期检查拓扑结构中所有服务器的状态。这个工作不会很复杂。例如，你可以建立一个调度任务，该任务可以启动 MySQL 客户端，并发出 `SHOW MASTER STATUS` 或 `SHOW SLAVE STATUS` 命令，然后打印或以邮件形式发送出结果。

密切监视服务器状态是发现潜在问题的好办法，它还使你有机会快速应对错误的发生。

你应该查找错误，监控滞后的 slave，检查过滤器以查看它们是否与记录相符，并确保所有的 slave 在运行且没有报告警告信息或连接错误。

## 检查日志

除了定期检查服务器状态外，还应该定期检查服务器的日志。实现方式是，使用 MySQL 工作台 GUI 依次连接每个服务器查看日志中的最新条目。

坚持不懈地执行此工作，将有利于前期的问题侦查。有时候某些错误或警告在其他地方可能都无法发现，但却被写入了日志。早点发现这些问题信号可以使修复工作更容易。

524

建议在查询服务器状态的同时检查日志信息。

## 检查配置

除了检查服务器状态和日志文件外，你还应该例行检查配置文件信息，以确保文档信息是最新的。该检查不像日志检查那么重要，往往容易被忽视。我们推荐至少每周检查一次配置文件，而且如果更新很多的话，还要更新文档，如果对拓扑结构或服务器的更改很少，建议每月更新一次文档。如果拓扑结构环境中不止一个管理员，可能需要考虑更加频繁地做上述工作。

525

## 有序地执行关闭操作

有时候，在诊断和修复问题时停止复制是必要的。如果复制已经被停止，可能不需要做任何事情，但是如果拓扑很复杂，而且错误与数据丢失相关，这时停止整个拓扑的复制可能更安全。但是应该以可控的和安全的方式停止复制。

以可控的方式关闭复制拓扑的策略有几个。如果数据丢失而 slave 又没有延迟，可能需要先锁住 master 上的表并刷新二进制日志，然后等待所有的（剩余的）slave 跟上，再关闭 slave。这保证了在正常运行的 slave 上所有事件都被复制和执行。

另一方面，如果问题比较严重，可能需要启动拓扑结构的叶分支上的 slave，并在拓扑结

构上执行你的工作，保持 master 的运行状态。然而，如果保持 master 在运行状态中（例如，不锁定所有的表），而且更新操作的负载很重，诊断和修复花费的时间将很长，那么在重启复制时，slave 将滞后于 master。如果你认为修复将花费很长时间，那么最好停止 master 上的更新。

如果你面临一个很困难的问题，或者是在没有任何征兆的情况下随机发生的（或者更糟糕）问题，可以考虑完全关闭复制。特别是只有一个 slave 遇到问题的时候。关闭复制能够在诊断问题时隔离服务器。

## 有序地执行故障后的重启操作

按序重启复制也是很重要的。最好在可控的情况下重启复制拓扑，如在单 master、单 slave 的拓扑结构上。即使你的拓扑结构很复杂，优先启动最基础的复制块进行测试，确保问题在所有服务器重启之前被修复。

在处理某些局限于单个服务器或一系列事件上的问题时，隔离操作是必要的。这也有利于启动单 master、单 slave 的结构。在这种情况下，如果你不能修复问题，从 MySQL 专家那获得帮助更容易些，因为你已经将问题缩小到尽可能小的参数集上。参看本章后面的“报告复制错误”一节获取更多的细节信息。

526

## 手动执行失败的查询

最经常忽略的一件事是为了找出错误线索而检查中继日志或二进制日志中的查询。在研究 slave 上的错误和诊断所有可能出错的事情的时候，很容易被困住，但是有时候（尤其是涉及查询操作的时候），隔离有问题的 slave 并尝试手动执行查询操作，可能会得到更多错误信息。

如果使用基于语句的复制，这将是件很容易做的任务，因为查询操作在二进制日志或中继日志中是可读的。如果使用基于行的复制，仍可以执行该查询操作，但是读不懂该查询本身。在这种情况下，一个畸形的或使用了不存在、不正确的或引用损坏的查询操作并不是那么易懂，只有手动执行了这个查询才能明白其意思。

记住，应该总是在试图诊断数据变更之前备份数据。运行坏查询是导致数据变更的一个常见例子。我们已经看到，有些导致复制错误的查询操作有时候手动执行可以成功。当这种情况发生时，通常表明 slave 的配置出错了，或二进制日志或中继日志出错了，而不是查询操作本身出错了。

## 不要混合使用事务型表和非事务型表

对很多人来说这是一个显而易见的最佳实践，但我们还是要提醒自己。在一个事务中混合使用事务型表和非事务型表会带来很多问题，这丝毫不逊于无法对非事务型表的变更进行回滚。应该彻底避免这个问题。

说是这么说，但还是有些混合的非事务型语句被认为是安全的。例如，如果非事务型表是查找表，很少被更新，更为重要的是，不在事务中更新，那么执行这些语句就可能是安全的。

解决混合使用事务型表和非事务型表的问题的最好办法是将所有表转换到 InnoDB 中，从而将它们都变成事务型的。大部分情况下，这是完全合理的，而且没有任何副作用。但是，如果数据要求使用不同的存储引擎（比如 Archive 或 CSA 存储引擎），可能就不方便了，甚至不可能进行转换了。

## 一般步骤

执行上述实践还有一些常用步骤，包括在 master 上复制的故障排除和暂停复制。

### 复制失败的故障排除

527

解决复制问题的步骤如下所示。其中大部分步骤已经在本书的其他地方介绍过，或者你可能已经对它们很熟悉了。

1. 检查 master 的状态，并记录其异常信息。
2. 检查 slave 的状态，并记录其异常信息。
3. 阅读所有服务器上的错误日志。
4. 检查每个服务器上的进程列表，查找状态中的异常信息。
5. 如果没有报告错误，则试图重启 slave，并记录出现的任何错误。
6. 检查有问题的服务器的配置信息，以确保没有什么被改变。
7. 检查笔记，并为研究和修复问题制订计划。

这个过程虽然过于简单化，但是它可以帮助你快速诊断复制问题，而不是纠结于具体错误（如果有错误的话）。



应该在执行这个过程的时候，记下所有的观察数据。在一个地方统一查看这些信息有时候可以更清晰地看到问题的全貌。

## 暂停复制

要暂停复制，在每个 master 上执行以下步骤，从主 master（第一个 master）开始：

在 master 上：

1. 运行 `FLUSH TABLES WITH READ LOCK` 命令。
2. 运行 `SHOW MASTER STATUS` 命令。
3. 记录 master 的二进制日志和位置。

在 slave 上：

1. 运行 `SELECT MASTER_POS_WAIT(binary_log_name, position)`。

在每个连接到 master 的仍保持活动的 slave 上，查看 `SHOW SLAVE STATUS` 命令的结果，直到 slave 与 master 同步。一旦 slave 达到同步的状态，这时关闭 slave 是安全的。

**528** 当你重启复制拓扑时，所有的 slave 将自动启动而没有延迟。这个过程在以下情况下是有用的：当系统发生损坏时，你希望在 slave 没有延迟的情况下恢复。

## 报告复制错误

有时候你可能会遇到一个无法解决或没有解决方案的复制问题。这时，应该向 MySQL 开发者报告问题，他们可能更能够诊断问题（如果是缺陷的话他们能够更正问题）。即使你没有适当的支持协议，但是可以使用 MySQL 进行故障报告。

这并不是说故障数据库是一个“免费帮助”资源，这适用于报告和跟踪缺陷。解决难题的最佳建议是首先查看所有的 MySQL 数据库论坛，寻找点子和解决方法。只有当问题是唯一的，并且与环境或应用程序无关的时候，才认为这是个缺陷（defect）。

最好的故障报告描述了可以通过独立的测试实例演示或重复的那些错误（例如，一个复制问题可以在测试 master 和 slave 上演示），而且该问题的演示只需要最少量数据集。通常这不过是关于有问题的配置和事件的完整准确的报告。

要报告故障，请访问 <http://bugs.mysql.com>，然后（如果你没有注册过）请在该网站注册



一个账户。当上传故障报告时，请一定尽量完整地描述问题。为了快速得到结果和解决办法，请在问题描述中尽量包含以下细节信息：

- 一字不差地记录所有的错误信息。
- 包含二进制日志（全部或摘录）。
- 包含中继日志（全部或摘录）。
- 记录 master 和 slave 的配置。
- 复制 SHOW MASTER STATUS 和 SHOW SLAVE STATUS 命令的结果信息。
- 提供所有错误日志的副本。



请务必使用上面介绍的所有技术，使用所有可用的工具探索问题。

只要故障报告被更新，MySQL 故障跟踪工具都将以邮件警告的方式发送故障报告。

529

## 小结

本章介绍了使用复制时遇到的常见问题的解决办法。探究了复制失败的原因，并讨论了如何解决这些问题，以及在将来如何预防这些问题。还介绍了 MySQL 复制的故障排除的工具和最佳实践。

通常被忽略的一个资源是在线参考手册中的“复制功能和问题”一节。这一节讲述了复制有限性的重要信息，如何与特定服务器功能和设置进行交互，以及数据库的设计和实现。在使用新版本服务器的时候阅读这一节，可确保你了解新功能对复制的影响。

听到老板向他的门口走过来，Joel 笑了，此刻他已经准备好了怎样向老板报告。一看到 Summerson 先生锃亮的皮鞋头，Joel 就开始说：“我已经恢复了西雅图的服务器，先生。是因为 slave 上的表被改变，从而导致某些查询失败而引起的问题。我已经建议西雅图那边将所有的模式更改和表维护都定向到这边的 master 上。从现在开始，我会留意观察，一出问题马上告诉您。”

Summerson 先生笑了笑。Joel 感觉到自己前额都流出了汗珠。“好，干得好。”

Joel 如释重负：“谢谢你，先生。”

老板正准备离开，却又停了下来，转过身，说：“还有件事情。”

“先生？”Joel 快速说道。

“是 Bob”。

Joel 有些疑惑。Summerson 先生的任务通常是一个完整的句子。“先生？”他犹豫地说。

“你可以叫我 Bob。我觉得你最好去掉‘先生’这个称呼。”

Joel 坐了一会儿，才意识到自己在盯着空白处发呆。他眨了眨眼，看到开发部门的好友 Amy 站在了他的办公桌前。

“你还好吗，Joel？”

“什么？”

“你在发呆，你在想什么呢？”

Joel 笑了，斜躺在座椅上。“我想我终于搞定他了。”

“谁？Summerson 先生？”

“是啊，他刚才让我喊他 Bob。”

Amy 笑了，交叉了胳膊，然后开玩笑地说：“嗯，我看我们以后要喊你 Joel 先生啦。”

Joel 笑着说：“吃过午饭了吗？”

# 保护你的资产

Joel 打开桌子中间的抽屉找钢笔，这时他从门缝中听到一个熟悉的声音：“Joel，我需要你与信息保障审核员（information assurance auditors）谈谈，告诉他们我们的恢复计划。还有，再订些磁带，好让他们知道我们已经做了准备。”

“审核员？磁带？”与 Summerson 先生目光相触时 Joel 问道。

“务必按照他们的要求开始进行所有灾难性恢复计划文档。还有，告诉采购人员你的文档需要什么类型的媒体。他们会告诉你填什么表格。再额外预备一些，以防万一。”Joel 的上司说道，接着就离开了。

Joel 关上抽屉后站起来，心想老板指的是备份介质。但是审核员和计划又是什么意思呢？Joel 将椅子拉到电脑前坐下。“嗯，我想我必须搞清楚什么是信息保障，以及如何进行备份。”他边打开浏览器边喃喃自语。

Summerson 先生把头伸到门口说道：“哦，还有，Joel，希望明天早上你能把恢复计划放到我的桌子上，在 14:00 的连续性业务规划会议之前。”

Joel 叹了一口气，他意识到老板需要的远远超过他的想象。Joel 顺手拿起了 MySQL 这本书。“这钱花得值。”然后翻开书中“保护你的资产”一章。

本章关注于数据保护和数据恢复，主要讨论备份计划、数据恢复和 MySQL 中备份和恢复的流程。本章没有对信息保护、信息的完整性、灾难恢复及相关概念进行深入介绍。事实上，一旦阅读了整个章节，你将学习到很多关于保护你的资产的知识，而不是仅仅知道备份和恢复数据。

# 什么是信息保护

本节介绍信息保障的概念、对组织的意义，以及如何进行良好的信息保护。这里不打算全面介绍所有内容，但会介绍一些基本知识，使你可以为学习该课题做好准备。对了，你还可以向老板介绍它呢。

信息保障（IA）是对信息系统技术的管理，旨在管理、监控、确保信息系统的可用性及安全控制等。IA 的目的是控制和保护数据和系统、访问权限，以及信息的机密性和可获得性。虽然本章主要针对数据库系统，但是 IA 适用于信息技术的所有方面。以下网站包含了有关 IA 使用的背景知识。

- NSA 网站中的 IA 章节：<http://www.nsa.gov/ia/>
- 维基百科中的 IA 定义：<http://bit.ly/wiki-ia>

## 信息保障的三个实践

有关研究人员将 IA 应用分为三类相关实践：

### 信息安全

通过内部和外部的管理手段来降低风险、减少威胁，从而保护计算机系统。

### 信息完整性

保证系统及其数据的持续可用性和连续性（有时被称为数据保护）。

### 信息价值

用户对系统的价值评估。

其中信息安全最常见且记录最完善。如今有无数参考资料是关于如何更好地维护计算机系统安全的，还有许多参考资料是关于物理安全措施方面的。

然而，信息完整性是这三者中最重要、但也最容易被忽略的。本章主要介绍信息完整性的最佳实践的重要性。

533 信息的价值通常是被了解得最少的应用。具有全面质量管理及相关学科的组织比较熟悉系统价值评估，但是一般只考虑企业的财政方面，而没有考虑与员工工作有关的信息的价值。尽管有人说，对公司有利就是对员工有利，但我们也遇到过这样的情形：有些看起来似乎用心良苦的政策，却可能阻挠了员工的工作。

在考虑信息完整性及其价值时，IA 还涉及灾难计划和灾难恢复，而它们往往被忽略。

良好的 IA 计划包括非物理的（数据、访问码等）和物理的（计算机、访问卡、钥匙等）



两方面。本章专注于物理方面，因为数据库系统可能是组织最重要的资源之一。

## 信息保障为什么重要

随着计算机系统对组织重要性的提升及技术成本的提高，节约花费和未雨绸缪成为优先考虑的因素。因此，IA 在研究机构和政府机构（最有兴趣的两个机构）之外的领域的应用也得到了推广。相关的概念越来越受欢迎，在很多公司的审核员的工具箱中找到了用武之地。

作为信息技术的领导者，你需要为企业不可预测的灾难做准备，其中包括物理的、非物理的财政威胁，保护企业免受灾难。一个常见的选择是使用 IA。如果你的组织已经或正在计划启动自己的 IA 计划，应该学习一下 IA 是如何影响你的。

## 信息完整性、灾难恢复及备份的职责

信息完整性有时又称业务连续性，保证组织在不中断的情况下完成任务，并且可以对一些意想不到的灾难性事件进行可控恢复。这种灾难性事件可以是一个小问题，也可能是操作或数据的毁灭性丢失，后者无法快速解决（如断电恢复）或者缺少资金（如更换信息技术系统）。重要的是，需要认识到没有单一的方法可以帮助你预防或恢复所有事件。



记住，硬件可能失效，最终也一定会失效，所以请提前做好准备。

534

信息完整性包括以下几个方面：

### 数据完整性

确保数据永远不丢失，且始终保持版本最新，不被损坏。

### 通信完整性

确保通信链路始终可用，且在中断的情况下可被恢复，可以启用一个可信连接。

### 系统完整性

确保在系统出现故障的情况下可以重启系统，在丢失或损坏的情况下可以恢复数据和服务器状态，保持系统的连续性。

随着业务越来越依赖于其信息系统，系统在业务运营中的作用也变得越来越重要。事实上，大多数使用信息技术的现代业务已经变得如此依赖这些系统，使得业务及其信息技术已合为一体（也就是说，业务无法脱离信息独立存在）。

如果这变成事实，组织必须认识到如果其信息系统失效，它的业务将不能有效运作（或者可能完全无法运作）。在这种情况下，企业需要尽快恢复系统。在发生突发事件时，恢复操作能力及数据，称为灾难恢复（disaster recovery）。

在监管机构审查下的企业应该知道（或者即将知道），一些法律决策和标准可能要求公司采用 IA 和灾难恢复措施。更重要的是，保护数据的工作已经取得一定进展。在美国，这些工作包括 1996 年的健康保险携带和责任法案（HIPAA）、爱国者法案和 2002 年的萨班斯 - 奥克斯利法案（SOX）（<http://soxlaw.com>）。

## 高可用性与灾难恢复

大多数企业认识到他们必须进行技术投资，使系统能够从轻微或中度事件中迅速恢复。这些技术包括复制、廉价磁盘冗余阵列（RAID）、冗余电源供应等。这些都是高可用性选项，因为它们允许在没有任何数据丢失（或者尽量将数据损失减小到最小）的情况下实时或近实时地恢复系统。

不幸的是，很少有企业采取额外的措施保护他们的资产免受毁灭性的（且昂贵的）损失。那些即使采取措施的企业也是采用了最终形式：直接使用灾难恢复。

在高可用性和灾难恢复之间确实存在重叠。高可用性解决办法可以解决小灾难，甚至可以作为主要灾难的防御层。但是两者提供的保护类型不同，高可用性抵御已知或可预知的事件，而灾难恢复可以解决意料之外的灾难。

## 灾难恢复

灾难恢复涉及处理过程、相关政策及灾难性事件后的信息完整性的规划。最重要的是创建和维护灾难恢复计划文档。

这里将介绍灾难恢复的各个方面，本节给出了这些方面的概述，这样你可以向你的管理团队说明灾难恢复的重要性。

第 3 章已经讨论了一种灾难恢复形式。这些有时候是组织机构用于恢复小型灾难的前期工具，但是如果情况真的很糟糕怎么办，比如 RAID 阵列出现了无法恢复的故障，或者服务器被烧毁？

在做好最坏的打算之前，你需要回答一系列问题，这些问题构成了灾难恢复计划的前提或目标，同时也是评价计划有效性的标准。

- 你的组织机构面临哪些物理的和非物理的威胁？
- 什么等级的操作能力能够维持你的业务需求？

- 你的业务在系统恢复之前能够等待多长时间？
- 可以利用哪些资源来计划和执行灾难恢复？

灾难恢复的第一步是认识到最坏情形：由于灾难性事件而导致的整个数据中心损失。不仅包括技术损失（服务器、工作站、网络设备等），还包括计算设施本身的损失，还有不要忘记操作系统的人力损失（这一点很少被考虑到，但是应该被考虑）。

虽然这看起来像是世界末日，但是现实中已经发生过并可能重复发生的事件可能会带来类似的破坏。它可以像大范围停电一样简单，也可能像飓风、地震甚至战争一样具有灾难性。

536

想象一下，信息技术所在的大厦由于严重的暴风雨而被损坏且无法修复。屋顶被撕裂，90%的物理资源被水和掉落的碎片损坏。再假设你的公司正在进行一项赚钱的业务交易，并且对时间要求敏感。

那么，你的公司将如何从这个灾难中恢复？对于该事件是否有任何计划？操作丢失将给公司收益带来多大的损失？这些操作多快可以恢复？这些都可能促使你的管理团队在灾难恢复上进行投资。

另外一个原因是，如果你做好最坏的打算，并拥有一套系统恢复流程，就更容易处理一般的中断。规划是灾难恢复的最重要组成部分，保证了业务的连续性。

## 灾难中无人幸免

像水灾或火灾这样的灾难，几乎可以在任何地点、任何组织中发生。有些地方更容易发生自然灾害。不论你的组织机构建在何方，总存在一些风险因素。但是，有一种类型的灾难可以在任何地点、任何时候发生：人为灾难。

所有的组织机构都需要防止恶意的人为活动，包括由内部人员执行的恶意操作。即使系统的某些部分得到了全面保护，有些部分还是可能会被少数不安全的站点干扰。

例如，假设你的公司所在地很少发生自然灾害，远离地震断裂带，也没有什么水灾风险。公司的建筑配有优良的灭火系统，并且保护良好，还有训练有素的应对人员进行全天候的监控。总之，由于自然灾害引起的物理性损失的风险很低。前人已经想好了一切，管理团队觉得不需要灾难恢复计划。

现在，假设某个雇员想从内部破坏组织。在设计灾难恢复计划的时候，问问自己：“一个被信任的员工会对系统、数据或基础设施造成什么样的破坏？”虽然这样看起来令人不安，甚至有些偏执，看看那些由内部的蓄意破坏产生的报告，便会让你觉得无法想象——你的雇员可能是破坏者。



良好的灾难恢复计划包括减少风险破坏的措施，这些措施包括识别潜在的漏洞，例如进入建筑和访问设备，以及查看敏感系统和数据的管理权限等。

灾难恢复的目标是尽快重建组织的运营能力，这意味着必须恢复其信息技术。灾难恢复的最佳实践包括风险评估和各种信息技术恢复的规划——包括物理技术和电子技术。良好的灾难规划包括重新确定组织的物理位置，以及重新构建信息技术。还要包括通过权宜之计获得任何设备的能力，这些方式包括使用备用设备、用其他站点的设备取代失败站点的设备或者采购新设备。

重建网络和数据需要使用最少的计算和网络设备，将数据和应用恢复到可接受的操作水平。因此，需要确定至少使用哪些技术，使得组织的运营收益损失较少或没有损失。

人员也是规划过程中必须考虑的关键资产之一。因此，在紧急事件发生的时候恢复程序会通知所有人员。可能只需要一系列简单的电话就可以办到，每个员工负责向一组同事传递重要信息。也可能是一个复杂的自动联系系统，向所有雇员发送预先记录好的信息。大多数自动系统通过员工的确认信息来进行评估。最好的通知系统还支持多点升级，例如，首先电话通知员工的家庭，接着是电子邮件、移动电话和短信息等。

灾难恢复还需要政策支持，包括紧急事件的风险评估（什么资产处于危险中，每种资产有多重要），以及如果领导层不在（或者去世）由谁来做决定。例如，如果网络负责人员不在，就指定其下属承担相应的责任（或进行必要的决策）。

## 灾难恢复规划

规划的内容很广泛，包括所有已知的或预期情景的应急方案。例如，一个良好的灾难恢复计划有一套书面程序说明如何重建站点，以及从头开始建立信息系统的步骤说明。

我们将指出某些灾难恢复规划的一个致命的缺陷。将大量的时间花费在资源和制订灾难恢复计划，然后存到系统硬盘上，这样做得不偿失。请花点时间保护好你的灾难恢复计划副本吧。

## 灾难恢复 workflow

目前你可能还有几个疑问，如何开始灾难恢复呢？怎样调查相关信息技术并交给老板一份灾难恢复计划？一切都从目标考查开始，从目标开始进行计划，最终就会形成自己的灾难恢复文档。

这听起来需要做大量工作，好吧，事实上确实如此。大多数组织机构成立了跨组织的专业团队。最成功的团队是由这样的人组成的：他们知道危机是什么（组织可能破产，且每个人都可能失去工作），并且他们的专业知识足以识别系统中的不稳定因素。



显然，你还必须做好这样的准备：那些建立、测试和制订灾难恢复计划的人可能不是该计划的执行人员。这需要注意以下重要的两点：全面而明确的文档，以及清晰的责任划分。

下面只是简单地描述了常见灾难恢复计划的步骤，后面的章节会给出详细介绍：

1. 建立你的灾难恢复小组。假设组织中有很多员工，那么现在就告诉经理：你需要一个团队来完成灾难恢复计划。确定该团队的核心角色，保证团队中包含各个部门的人员，避免只有部分技术熟练的人。你需要纵览整个组织机构，只有使团队成员多样化才能达到这个目的。
2. 制订任务声明。研究系统的当前操作状态，明确组织可接受的最低操作水平，确定灾难恢复过程的目标。
3. 让管理层介入。如果该任务不是由管理层发起的，否则你需要说服你的管理层加入，因为灾难恢复需要他们的支持才能成功。这可能是获得所需资源(时间、预算、人员等)完成目标的唯一方法。
4. 为最坏的结果做准备。很多人觉得这一条最可笑。你应该记录以下情况：本地区可能发生的灾难性事件的类型，包括所有来自于盗窃、蓄意破坏、气候事件、火灾，甚至更糟的灾难事件。务必从这些情况开始记录文档，因为后面开发恢复计划时需要用到这些信息。
5. 评估库存及组织资源。为组织的所有信息技术列一个清单。务必包含系统成功运作所需的所有技术，其中包括员工工作站和网络连接。你应该列出目前存在的东西，而不是你认为的最小集合。将资产减少到最小集合是后面需要做的事情。还需要记录目前的组织结构图和指挥链，并标出主要和次要决策者。所有这些都要记录并保存。
6. 进行风险评估。确定每一项资源损失可能对组织造成的影响，这将形成系统的最低操作水平。应该为每个资源设定优先级，甚至建立多个可接受的操作级别。一定要谨慎研究应用程序及其需要的数据，这将有助于决定事件响应的程序。你可能只需要部分恢复，例如，为了节约时间和资金，只是重新部署剩余的系统，而不是建立信息完整性、灾难恢复及备份的职责等一个全新的系统来进行灾难恢复。这样你可以在较低的性能下短时运营。
7. 制订应急计划。你已经了解了灾难场景，熟悉了库存资产，并进行了风险评估，现在可以开始起草灾难恢复计划了。应急计划可以采取任何你觉得有用的形式，但是大多数计划者采用列表和叙述的形式制订流程。灾难计划书写格式不限，只要被团队认可就行。

539

8. 建立验证程序。一旦计划在执行过程中失败，那么计划将毫无用处。现在需要制订灾难恢复计划的验证程序。从为每个应急事件制订测试实例开始，然后执行必要的调整。回到步骤 4，循环后面的步骤，直到完善了灾难恢复计划为止。
9. 熟能生巧。精炼灾难恢复计划之后，开始实战练习，以确保计划可行。最终的目的是向管理层展示可以实现各个级别的运营水平。否则，回到步骤 6，循环后面的步骤，直到得到一套完整且可重复的流程。

灾难恢复计划应该每年至少进行一次完整的操作测试，或者在组织或信息技术发生重要变化时做完整的操作测试。

## 540 灾难恢复工具

虽然灾难恢复不仅仅包括数据和计算机，但是灾难恢复的重点常常是数据和信息技术能力。有很多工具和策略可以用于灾难恢复计划。下面介绍一些常见的工具和策略。

### 备用电源

通常包括某些不间断供电形式。这类设备的成本和耐用性在很大程度上取决于公司允许的停机时间。如果要求必须连续运营，则需要一套可以长时间为设备供电的系统。

### 网络连接

根据你的需要通过电子方式与客户和业务伙伴交流，你可能需要考虑冗余的或其他替代的网络访问。这可能简单到仅通过使用不同的媒体（如再加一条连接到访问点的光纤）来实现，也可能复杂到需要使用其他替代的连接点（如卫星或移动设备）。

### 替代站点

如果你的公司必须具有完整的能力且希望尽可能短时间的停机，你可能需要确保在没有任何数据或操作丢失的情况下迅速恢复系统。最后的办法是使用替代站点安置你的信息技术甚至整个公司。这个替代站点形成一个移动办公室（临时办公室拖车或类似平台），它包含了你的信息技术的副本。

### 替补人员

灾难恢复计划常常忽视这部分。一个必须考虑的可能的灾难是损失部分甚至全部关键工作人员，例如，传染病、被敌对公司挖走甚至是死亡。不论何种情况，都应该对组织机构内的重要角色进行分析，并提出灾难事件中的替补措施。显然可以选择交叉培训员工。这样可以丰富你的人才库，还可以提高员工价值。

### 备份硬件

连续运营最明显的需求就是额外的硬件。高可用性技术可以部分满足这种需求，但

最终还是要在站点外（或在替换站点上）存储备用硬件，这样才能将它们快速地投入服务中。这种硬件绝不能只是摆设，而应该保持更新并定期运行。

### 安全库

计划者容易忽视的另一个方面是在安全的地方存储重要数据的备份。应该准备一个安全的存储空间（安全级别取决于你的数据的价值和敏感度）以匹配你的停机需求。也就是说，如果要求停机时间很短，那么银行或类似的库可能无法快速获取数据，而将关键数据备份存储在地下室或公共场所又可能不够安全。

### 高可用性

如前所述，高可用性选项形成了灾难恢复计划的第一层防御。将失败转移到其他数据存储库可以帮助你应对一些小故障。



始终将你的数据和灾难恢复计划的副本保存在站点外的安全位置。

所有灾难恢复工具和策略的一个经验法则是：需要从灾难中恢复的速度越快，需要恢复的操作能力越强，那么你付出的成本将越高。这也是需要开发具备多个级别的运行能力的系统很重要的一个原因——那样你可以调整计划，并且可以在不同层次的损失和环境下运营。有可能你的公司遭遇经济衰退或其他财政限制，不得不牺牲某些运行能力。考虑到的情况越多，灾难恢复计划就越强大。

## 数据恢复的重要性

大多数企业把数据看成是最有价值的资产（超过员工的价值，也是最不容易被代替的资产），所以大多数灾难恢复计划的关注点是恢复数据，并使这些数据可用。这些都是数据问题。

数据恢复是指将组织机构的数据恢复到可接受的运行状态的一系列计划、原则和流程。数据恢复最重要的就是数据的备份和恢复能力。成功的备份和恢复能力有时也被称为弹性（resiliency）或可恢复性（recoverability）。

数据恢复主要是计划、执行和审计数据备份，以及恢复数据操作。其中审计经常被忽略。一个优秀的备份和恢复服务应该是可靠的。通常某些地方出问题，尝试恢复的时候，发现备份系统失效，或数据不存在，或归档损坏，甚至被破坏，之前没有任何通知。显然，如果数据不存在或在发生灾难时被损坏，数据恢复起来会很麻烦。



## 术语

几乎所有的数据恢复和备份解决方案都会用到这两个术语，所以了解它们是很重要的：

### RPO (Recovery Point Objective)

系统运营可接受的状态。因此，RPO 代表了服务（运行能力）下降的最大可接受程度或数据丢失的最大可接受程度。使用这些标准可衡量恢复操作是否成功。

### RTO (Recovery Time Objective)

能够容忍系统不可用的最长时间，又称停机时间。

RTO 在很大程度上受 RPO 影响。如果需要恢复所有数据（或接近所有数据，例如，复制过程中的所有事务），则恢复过程会麻烦些，因而越想做到快速恢复，即相对较低 RTO，则需要付出的成本将越高。RTO 较低且 RPO 较高，由于这样的系统通常都很复杂，所以要达到较高的 RPO 或较低的 RTO 需要更高的软硬件成本，还需要针对该系统对员工进行培训。例如，如果想要 RTO 低于 3 秒且 RPO 能够保证无数据丢失，则必须制订一个可扩展的高可用性解决方案，该方案可以容忍在服务器及数据存储等完全丢失的情况下，无数据（或时间）丢失。

## 备份和恢复

本节将介绍两种重要的数据恢复工具，还将介绍备份和恢复的概念、需要执行的计划，以及备份和恢复的解决方案。

对于像 MySQL 这样的数据库系统，备份和恢复功能是指复制数据副本，存储这些副本然后重新加载，使数据回到备份时间点的状态。

下面向不太熟悉备份和恢复系统及 MySQL 的可行方案的读者进行相关介绍。我们将探索各种解决方案的好处，以及如何创建存档计划，使得恢复数据的损失最小。

### 为什么要备份

有些人可能认为如果使用了复制或某种形式的硬件冗余，就没有必要进行备份了。虽然

543

对于机械或电子故障事件中数据的快速恢复来说，这种说法或许是合理的，但是如果发生灾难性事件，这些措施将无济于事。

表 15-1 描述了最有可能丢失数据的情况，并给出了各种情况的恢复方法。可以看到，大部分情况都或多或少地受益于备份。除了可以从失败和错误中进行恢复以外，还可以将备份纳入日常数据保护计划中。这么做的理由很充分，你可以为复制拓扑添加 slave 服务器，将其用作故障转移工具，甚至作为网络之间传输数据的一种方式。



表15-1: 常见的数据丢失原因

故障类型	描述	恢复方法
用户错误	用户无意删除数据或将数据更新为不正确的值	从备份中恢复删除的数据
断电	一个或多个系统断电	使用连续供电系统
硬件故障	一个或多个系统或组件发生故障	使用冗余系统或复制数据
软件故障	数据在传输过程中被改变或丢失	该类故障可能很难侦查, 需要从备份中恢复数据 (如果没有办法修复转换操作)
基础设施问题	放置设备的基础设施无法使用, 且数据连接丢失	可能需要一套新的基础设施, 用于建立一个可以接受的运行系统
网络故障	装载数据的服务不能访问	重新连接或建立一个新的数据储存库
蓄意破坏	数据被有意偷走或损坏	关闭安全漏洞, 并审查数据

如果开发了自己的日常备份计划并使用了异地存储, 那么你可能已经做好了足够的灾难恢复准备。例如, 假设数据库系统的冗余硬件, 甚至复制服务器 (即 MySQL 复制中的 slave) 突然消失, 或者被盗, 或者发生了灾难事故, 这时将发生什么? 异地站点的定期备份允许将数据或数据库系统恢复到最后一次备份的状态。

现在假设你的数据库系统运行在商业硬件上 (MySQL 因独立于硬件而出名)。要想实现备份, 你可以购置新硬件或重用已有硬件, 使数据库系统迅速恢复运行 (当然, 这取决于需要恢复的数据量)。

## 硬件恐怖事故

表 15-1 描述了数据丢失的常见分类。硬件故障不经常发生, 但是一旦发生远比表 15-1 中描述的严重得多。下面给出现实中可能发生的硬件故障, 以及相应的恢复办法。

### 丢失硬件阵列中的多个磁盘

如果一个硬件 RAID 的多个磁盘发生故障, 很可能无法恢复。如果发生这种情况 (发生的可能性往往比你想象的大), 那么别无选择, 只有从备份阵列中恢复数据。

### master 和 slave 上的磁盘都发生故障

如果你的 master 和 slave 系统都发生故障 (尤其是发生磁盘故障), 那么你将丢失热备份 (hot standby)。这种情况有多种表现形式, 例如, 某个特定的表 (或分片, shard) 或者其所在的磁盘可能同时在 master 和 slave 上被损坏。这时恢复数据可能是唯一的解决办法。

### 备用电池发生故障

这也许是最丢人的状况。高效的供电备份系统在你需要它的时候 (在停电时期) 发生了故障。如果你需要为停电做准备, 请考虑为主电源备份故障建立一个备份计划。

544

关键在于，日常数据备份是一项基本且完备的数据恢复实践。高可用性如复制或 RAID 硬件对秒级恢复来说显然更好，但是它们无法处理灾难性损失。我们已经知道复制可以防止数据丢失，但是如果 master 和 slave 都损坏了且不可修复，那又该怎么办呢？在这种情况下，只有有效的最新备份可以拯救你。

以下各节将详细讨论备份，并展示如何创建 MySQL 数据的备份。

## 备份的期望

备份以某种可恢复的方式创建数据的副本。此外，备份副本必须保持一致。对于事务性数据库，这意味着备份只包含备份数据之前提交的事务，不包括部分提交或未提交的数据。备份还应该支持监控，用于验证备份的性能和数据状态。

有以下几种数据备份形式：

### 完全备份

对服务器进行完整备份，没有任何遗漏。该种形式周期最长，占用的存储空间最大。

545

### 差异备份

仅对上一次完全备份之后发生变化的数据进行备份。一般来说，该种备份比完全备份需要的空间少，而且速度也较快。

### 增量备份

只备份自上次增量备份或完全备份后发生变化的数据。通常需要某种形式的更改日志（如 MySQL 中的二进制日志）。该类备份通常比差异备份或完全备份周期更短，另外，根据自上次备份以来发生的变化量的大小，所需的备份空间可能很少。

差异备份和增量备份都是备份当前和上一次状态的差异数据。但是差异备份包括真实数据，但增量备份只是备份从中可以恢复数据的日志。因此当你从一个差异备份中恢复数据时，差异数据需要恢复到一个完整备份数据库中。当使用增量备份时，数据库则需要从日志中恢复数据。

制订数据恢复计划时，需要为备份文档（也称为备份映像或备份文件）命名。例如，用日期加上其他相关信息为文件命名，如 *full\_backup\_2013\_09\_09* 或 *incr\_backup\_week\_3\_september*。命名没有什么限制，可以根据需要使用任何你认为有意义的名字。

## 还原过程的期望

还原操作用备份文档中的数据替换系统中的数据，使得被替换的数据和文档中的数据相同。像备份一样，还原进程必须支持监控，以验证还原的性能和数据的状态。

不幸的是，很少有备份系统完全满足这些备份和恢复的准则。满足条件的通常是那些专业平台（使用定制的硬件和软件），这样的平台价格昂贵且很难维护。本章将介绍备份和还原 MySQL 数据的经济方案。

## 逻辑备份与物理备份

一个容易被误解的备份概念是逻辑备份和物理备份之间的区别。所选择的备份模式可能影响备份和恢复数据的效率。

逻辑备份（logical backup）仅仅是一些普通 SQL SELECT 查询的集合。通常通过表扫描，即遍历每条记录创建逻辑备份。

物理备份（physical backup）是指原始二进制数据（文件）的副本，这些副本通常是操作系统级别的文件。任何复制数据、索引和缓冲内存（文件）且不是逐条记录访问的备份方法都为物理备份。

546

可以想象，逻辑备份比物理备份慢得多。这是因为系统必须使用普通的 SQL 内部机制一次读取一个记录，而物理备份通常使用操作系统功能，没有那么多开销。但是，物理备份可能需要锁定表及其相关的二进制文件，直到整个复制过程结束；而某些逻辑备份在运行时不会锁定表或阻塞表的访问。

选择使用逻辑备份还是物理备份比你想象的要难。例如，如果想要创建包含所有数据的 MySQL 数据库服务器的副本，可以简单地将数据库离线，并关闭服务器，然后将整个 *mysql* 目录复制到另外一台计算机上。如果更改了默认路径，还必须复制 InnoDB 文件的数据目录。这样就建立了与第一个实例相同的服务器实例。这可能是建立复制拓扑的好方法，但是对于那些无法离线的重要数据库的备份来说，这样会很不方便。此时，逻辑备份可能是最好的（而且是唯一的）选择。然而，如果你的数据库存储在 InnoDB 中，那么相比逻辑备份而言，也许 MEB 是更好的选择，因为在备份 InnoDB 的时候不需要中止该数据库的活动。

## 制订归档计划

使用备份和还原工具需要遵循一定的原则，还有一些注意事项。最容易忽略的步骤是制订归档计划，这将决定你备份数据的频率。

首先问问自己：“我可以承受多少数据损失，然后从头恢复它们？”也就是说，如果数据全部丢失，你最多可以忽略多少数据？或者说你能够承受多少数据无法恢复？这就是你的恢复点目标（RPO），即进行业务运营必备的操作能力水平。

很明显，任何数据丢失都是坏事。你必须在规划 RPO 时考虑数据本身的价值。你可能发



现某些数据比其他数据更有价值。某些数据可能对公司的福利和稳定很重要，而其他数据或许没那么重要。显然，如果你确定某些数据对组织机构很重要，那么必须保证它们能够完全恢复。因此，你可能会想出若干不同的 RPO。一般情况下，最好先制订几个级别的 RPO，然后确定哪个更适合恢复需求。

制订归档计划很重要，因为它决定了备份频率及备份能力（即备份数据的多少）。如果你不能容忍任何数据丢失，更加需要扩展高可用性选项以支持异地备份某些数据。但是即使那样做也不能保证万无一失。例如，master 中的损坏或蓄意破坏可能波及其他副本，并在一段时间内未被发现。

通常采用恢复丢失数据所能容忍的最大时间来衡量对数据丢失的容忍程度。也就是说，在数据恢复过程中，你可以承受数据多长时间不可用？这直接说明了停机过程中你的组织机构可以承受丢失多少钱。对于有些组织机构来说，不能访问数据的每一秒钟的代价都是很高的。另外，某些数据比其他数据对公司赢利更有价值，这些数据的恢复需要花费更多的时间。

除此以外，还应该制订一些时间期限，根据当前业务的状态决定使用哪个时间期限，这些时间期限形成了恢复时间目标（RTO），与相应的恢复点目标（RPO）级别匹配，就可以知道不同时间期限的恢复过程需要多长时间。确定 RTO 级别需要将数据重新导入系统，或者重做某些工作以重新获得数据（例如，再次下载数据或从商业伙伴那里获取更新数据）。

确定了能够容忍的数据损失量（RPO）及其恢复时间（RTO）后，检查备份系统的能力，并选择能够满足需求的备份频率和方法。但还不止这些，还应该建立自动化任务，使这些任务在最有益（或损失最小）的时候自动执行备份。

最后，通过试验数据恢复（如还原数据）定期测试备份，确保备份的完备无缺和可实施性，从而保证在低风险下进行安全的数据恢复。

## 备份实用程序和操作系统层的解决方案

Oracle 提供了备份和恢复的多种方案，其中包含从高级商业物理备份（MySQL Enterprise Backup）到 GPL 逻辑备份（MySQL 实用工具和 mysqldump）。此外，还有很多第三方方案用于执行 MySQL 上的备份。你也可以在操作系统级别执行一些不成熟的但是有效的备份。

本章最后一节将简要讨论其他流行的备份解决方案，这些备份方案也十分有效，大多与下列相似：

- MySQL 企业备份



- MySQL 实用工具
- 物理文件复制
- *mysqldump* 工具
- LVM 快照
- Xtrabackup

## MySQL 企业备份

如果你在找一个物理级的、非阻塞的，且使用 InnoDB 存储引擎作为主数据存储的数据库备份方案，那么来自 Oracle 官方的 MySQL 企业备份可能是你需要的。MySQL 企业备份是 MySQL 企业版本中的一部分，更多关于 MySQL 企业备份的信息可以从 MySQL 企业官方文档获取 (<http://bit.ly/ent-prods>)。

MySQL 企业备份，简称为 MEB，可以为你的数据做在线全量备份，对数据访问无影响。MEB 也支持增量备份和部分备份，当恢复数据时，MEB 支持基于时间点的恢复 (RITP)，可以恢复到指定的时间点。当与二进制日志特性结合时，MEB 提供前滚恢复到一个指定的事务，通过备份可以提供部分恢复的功能。你也可以使用可传输的表空间（在 5.6 以后的版本支持）以方便部分恢复，但是需要开启 `--innodb_file_per_table=ON` 选项。

MEB 两个其他有价值的功能是特定表的执行或表空间的部分恢复，以及创建压缩备份，从而大大减少备份的大小。



MEB 同样可以备份 MyISAM、合并、分区和归档表，但会在备份的时候阻止对表的更新。

MEB 支持 MySQL 4.0 以上的各种平台版本，包括 Linux、Mac OS X 和 Windows。如果你有兴趣了解更多关于 MySQL 企业备份的知识，请访问 MySQL 企业备份文件的网站 (<http://bit.ly/ent-backup>)。该 MEB 工具有许多额外的功能，很多功能对于使用 Oracle 备份工具的用户会很熟悉。我们这里列出了几个有趣和重要的属性：

- 与外部备份产品集成（例如，Oracle 安全备份、NetBackup 和 Tivoli 存储管理器）使用串行备份磁带 (SBT)。这可以帮助你的 MySQL 备份集成到现有的企业备份和恢复解决方案中。
- 备份到一个单一的文件，该文件允许使用管道流，并且还使用 SBT。
- 并行备份允许分割的每个文件被备份到一个单独的缓冲器。这个缓冲器可以并行

用来处理读、写和压缩操作。这个特性可以提高备份效率 10 倍。注意，备份序列的完整性是通过多线程维护的。换句话说，仍然可以恢复文件而无须使用特殊的选项或程序将这些流汇总到一起，MEB 为你提供所有你需要的。

- 在最新版本中，MEB 做的是全实例备份。这意味着所有数据文件、日志文件，以及全局变量和插件配置的状态都被保存在备份映像中。酷！如果检查你的备份目录中的元数据目录，会发现 *backup\_variables.txt* 中列出了所有的文件，*backup\_create.xml* 中包含备份操作的设置清单。
- 支持新版本服务器的所有功能，包括使用不同大小的 InnoDB 的页面、不同 InnoDB 的 CRC 算法，以及在 SSD 上的 InnoDB 页的重定位优化。
- 与 MySQL Workbench 的集成。通过 MySQL Workbench 企业版调用。



你可以通过 Oracle 的 eDelivery 系统 (<http://edelivery.oracle.com>) 下载 MySQL 企业版备份的试用版进行评估，必须拥有一个 Oracle 网络账户。如果没有这个账户，可以在网站上新建一个。

很容易就可以使用 *mysqlbackup*，你可以通过指定命令名作为参数之一来发起一些命令。可用的命令如下，由基本命令组成（参见在线 MEB 文档可以获取每个命令的更多细节，同时可以获取相关的例子了解如何使用它们）。

#### 备份命令

##### backup

创建一个备份到指定目录中。由此产生的文件包含一个原始的备份映像，且在其可以被恢复之前必须确保日志可用。

##### backup-and-apply-log

创建一个备份到指定目录中，且在快照中使用日志操作，这样可以做到完整恢复。注意，执行该命令的结果文件不能被压缩，且它通常比使用 *backup* 命令消耗的空间更多。

550

##### backup-dir-to-image

在指定的备份文件夹中创建单一文件的备份（即映像）。

##### backup-to-image

创建备份到单一文件中。结果文件是一个原始的备份映像，恢复前需要应用日志。

## 还原、恢复命令

### apply-incremental-backup

该操作使用增量备份，并将其应用到一个全量备份中，然后再开始还原。注意，这个只用于 InnoDB 表。

### apply-log

该操作使用一个现有的备份目录，同时应用 InnoDB 的日志，以便将所得的 InnoDB 数据和日志文件用正确的大小将数据还原到服务器。需要注意的是，为了将日志应用到备份映像文件中，必须先将其解压缩到一个备份目录。

### copy-back

这是一个恢复命令。在这种情况下，服务器离线，这个操作复制所有的文件到指定的数据目录中，这是离线恢复。需要注意的是，你永远不应该在一个运行着的服务器上执行该命令。还需要注意的是，这并不应用日志。

### copy-back-and-apply-log

这是一个恢复命令。在这种情况下，服务器离线，这个操作复制所有的文件到指定的数据目录中，同时应用日志。这是一个完整的离线恢复。需要注意的是，你永远不应该在一个运行着的服务器上执行该命令。

## 工具命令 ( 管理映像 )

### extract

使用相对路径从备份映像中提取对象。--backup-dir 选项指定一个目录用于放置被提取出来的文件。也可以使用 --src-entry 选项来过滤特定文件或目录。

### image-to-backup-dir

抽取备份映像到备份目录中。

### list-image

列出映像中的对象。--src-entry 选项可以用于过滤特定文件或目录。

### validate

通过检查校验码确定文件是否损坏。

## MEB 如何工作

正如之前所说，MEB 是物理级别的备份工具。正因如此，它在磁盘上保存的文件是二进制副本。它也可以直接复制 InnoDB 引擎文件和日志。最后，MEB 也对其他相关文件进行备份，例如 .frm、MyISAM 文件等。随着 MEB 3.9.0 版本发布，可以对全实例进行备份，

意味着可以保存全局设置变量。

因为 MEB 复制 InnoDB 文件，所以也可以实现快照操作（因此也可以进行热备）。在这种情况下，InnoDB 存储引擎内部指针可能与当前备份映像数据不一致。因此，备份需要两个步骤；首先备份数据，然后应用日志到备份目录中的 InnoDB 文件上，以便它们被设置为正确的指针。换句话说，在这个过程中你需要让任何更改应用到所有的 InnoDB 表上后，备份文件与 InnoDB 文件的状态保持一致。做完这些后，备份映像状态被恢复。幸运的是，可以通过使用 `backup-and-apply-log` 命令（在下一节中描述）在一次操作中完成这两步。

恢复一个全量备份需要服务器离线。因为备份会用备份时的状态覆盖所有的文件和全局变量（MEB 3.9.0 和以后版本）。

以下各小节展示了三个基本操作：全量备份、增量备份和完全恢复。你可以在 MEB 在线参考手册中找到所有命令的例子（<http://bit.ly/ent-prods>）。

## 执行全量备份

开始创建全量备份，需要指定用户、密码和端口等选项（`mysqlbackup` 与标准 MySQL 客户端一样可以通过主机、端口或 socket 来访问）。还需要指定你选择的备份命令（这里我们使用 `backup-and-apply-log` 选项）以及存储备份结果的目录：

```
$ sudo ./mysqlbackup backup-and-apply-log --port=3306 \
--backup-dir=./initial-backup --user=root --password --show-progress
Password:
MySQL Enterprise Backup version 3.9.0 [2013/08/23]
Copyright (c) 2003, 2013, Oracle and/or its affiliates. All Rights Reserved.
```

```
mysqlbackup: INFO: Starting with following command line ...
./mysqlbackup backup --port=3306 --backup-dir=./initial-backup --user=root
--password --show-progress
```

```
Enter password:
mysqlbackup: INFO: MySQL server version is '5.5.23-log'.
mysqlbackup: INFO: Got server configuration information from running server.
```

**552** **IMPORTANT:** Please check that `mysqlbackup` run completes successfully.  
At the end of a successful 'backup' run `mysqlbackup`  
prints "mysqlbackup completed OK!".

```
130826 15:48:29 mysqlbackup: INFO: MEB logfile created at
/backups/initial-backup/meta/MEB_2013-08-26.15-48-29_backup.log
```



---

### Server Repository Options:

---

```
datadir = /usr/local/mysql/data/
innodb_data_home_dir =
innodb_data_file_path = ibdata1:10M:autoextend
innodb_log_group_home_dir = /usr/local/mysql/data/
innodb_log_files_in_group = 2
innodb_log_file_size = 5242880
innodb_page_size = Null
innodb_checksum_algorithm = innodb
```

---

### Backup Config Options:

---

```
datadir = /backups/initial-backup/datadir
innodb_data_home_dir = /backups/initial-backup/datadir
innodb_data_file_path = ibdata1:10M:autoextend
innodb_log_group_home_dir = /backups/initial-backup/datadir
innodb_log_files_in_group = 2
innodb_log_file_size = 5242880
innodb_page_size = 16384
innodb_checksum_algorithm = innodb
```

mysqlbackup: INFO: Unique generated backup id for this is 13775524146819070

mysqlbackup: INFO: Creating 14 buffers each of size 16777216.

130826 17:26:56 mysqlbackup: INFO: Full Backup operation starts with following  
threads 1 read-threads 6 process-threads 1 write-threads

130826 17:26:56 mysqlbackup: INFO: System tablespace file format is Antelope.

130826 17:26:56 mysqlbackup: INFO: Starting to copy all innodb files...

130826 17:26:56 mysqlbackup: INFO: Progress: 0 of 27 MB; state: Copying system  
tablespace

130826 17:26:56 mysqlbackup: INFO: Copying /usr/local/mysql/data/ibdata1  
(Antelope file format).

130826 17:26:56 mysqlbackup: INFO: Found checkpoint at lsn 20121160.

130826 17:26:56 mysqlbackup: INFO: Starting log scan from lsn 20121088.

130826 17:26:56 mysqlbackup: INFO: Copying log...

130826 17:26:56 mysqlbackup: INFO: Log copied, lsn 20121160.

130826 17:26:56 mysqlbackup: INFO: Completing the copy of innodb files.

130826 17:26:57 mysqlbackup: INFO: Preparing to lock tables: Connected to mysqld  
server.

130826 17:26:57 mysqlbackup: INFO: Starting to lock all the tables...

130826 17:26:57 mysqlbackup: INFO: All tables are locked and flushed to disk

130826 17:26:57 mysqlbackup: INFO: Opening backup source directory

'/usr/local/mysql/data/'

```

130826 17:26:57 mysqlbackup: INFO: Starting to backup all non-innodb files in
subdirectories of '/usr/local/mysql/data/'
130826 17:26:57 mysqlbackup: INFO: Copying the database directory 'menagerie'
130826 17:26:57 mysqlbackup: INFO: Copying the database directory
'my_sensor_network'
130826 17:26:57 mysqlbackup: INFO: Copying the database directory 'mysql'
130826 17:26:57 mysqlbackup: INFO: Copying the database directory 'newschema'
130826 17:26:57 mysqlbackup: INFO: Copying the database directory
'performance_schema'
130826 17:26:57 mysqlbackup: INFO: Copying the database directory 'test_arduino'
130826 17:26:57 mysqlbackup: INFO: Copying the database directory 'world'
130826 17:26:57 mysqlbackup: INFO: Copying the database directory 'world_innodb'
130826 17:26:57 mysqlbackup: INFO: Completing the copy of all non-innodb files.
130826 17:26:58 mysqlbackup: INFO: Progress: 27 of 27 MB; state:
Copying metadata to image
130826 17:26:58 mysqlbackup: INFO: A copied database page was modified
at 20121160.
(This is the highest lsn found on page)
Scanned log up to lsn 20123788.
Was able to parse the log up to lsn 20123788.
Maximum page number for a log record 776
130826 17:26:58 mysqlbackup: INFO: All tables unlocked
130826 17:26:58 mysqlbackup: INFO: All MySQL tables were locked for
0.913 seconds.
130826 17:26:58 mysqlbackup: INFO: Reading all global variables from the server.
130826 17:26:58 mysqlbackup: INFO: Completed reading of all global variables
from the server.
130826 17:26:58 mysqlbackup: INFO: Creating server config files server-my.cnf
and server-all.cnf in /Users/cbell/source/meb-3.9.0-osx10.6-universal/bin/
initial-backup
130826 17:26:58 mysqlbackup: INFO: Full Backup operation completed successfully.
130826 17:26:58 mysqlbackup: INFO: Backup created in directory
'/Users/cbell/source/meb-3.9.0-osx10.6-universal/bin/initial-backup'
130826 17:26:58 mysqlbackup: INFO: MySQL binlog position:
filename mysql-bin.000026, position 20734

```

#### ----- Parameters Summary -----

```

Start LSN          : 20121088
End LSN            : 20123788

```

```

mysqlbackup: INFO: Creating 14 buffers each of size 65536.

```

```

130826 17:26:58 mysqlbackup: INFO: Apply-log operation starts with following
threads 1 read-threads 1 process-threads
130826 17:26:58 mysqlbackup: INFO: ibbackup_logfile's creation parameters:
    start lsn 20121088, end lsn 20123788,
    start checkpoint 20121160.
mysqlbackup: INFO: InnoDB: Starting an apply batch of log records
to the database...
InnoDB: Progress in percent: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
mysqlbackup: INFO: InnoDB: Setting log file size to 5242880
mysqlbackup: INFO: InnoDB: Setting log file size to 5242880
130826 17:26:58 mysqlbackup: INFO: We were able to parse ibbackup_logfile up to
    lsn 20123788.
mysqlbackup: INFO: Last MySQL binlog file position 0 20653,
file name ./mysql-bin.000026
130826 17:26:58 mysqlbackup: INFO: The first datafile is
'/backups/initial-backup/datadir/ibdata1'
    and the new created log files are at '/backups/initial-backup/datadir'
130826 17:26:59 mysqlbackup: INFO: Progress: 27 of 27 MB; state: Completed
130826 17:26:59 mysqlbackup: INFO: Apply-log operation completed successfully.
130826 17:26:59 mysqlbackup: INFO: Full backup prepared for
recovery successfully.

```

mysqlbackup completed OK!



可以使用 `--show-progress` 选项来跟踪备份过程，会输出到标准输出、文件或表中。该选项还可以显示执行进度。可以通过在线 MEB 参考手册了解更多详情。

可以通过 `--compress` 选项来完成对备份的压缩，但是不能与 `backup-and-apply-log` 一起。要创建一个可以恢复的压缩的完整备份，需要做两步，如下所示（请注意，需要使用 `--uncompress` 选项和 `apply-log` 命令）：

```

$ sudo ./mysqlbackup backup --compress --port=3306 \
  --backup-dir=/backups/initial-backup --user=root \
  --password --show-progress
$ sudo ./mysqlbackup apply-log --port=3306 \
  --backup-dir=/backups/initial-backup --user=root \
  --password --show-progress --uncompress

```

## 执行增量备份

执行增量备份操作需要基于一个已有的备份映像（完全或者是前面的增量备份），与当前数据库状态进行比较，可使用 `--incremental-base` 选项指定基础备份目录。使用 `--backup-dir` 可指定增量备份目录。这里我们将演示如何从最近全量备份创建第一个增量备份。（请参阅在线 MEB 参考手册中通过指定逻辑系列号（LSN），手动执行增量备份的方法）：

```
$ sudo ./mysqlbackup backup --port=3306 \  
  --backup-dir=/backups/incremental-backup-Monday \  
  --incremental-base=dir:/backups/initial-backup-1 --user=root --password  
MySQL Enterprise Backup version 3.9.0 [2013/08/23]  
Copyright (c) 2003, 2013, Oracle and/or its affiliates. All Rights Reserved.
```

```
mysqlbackup: INFO: Starting with following command line ...
```

```
./mysqlbackup backup --port=3306  
  --backup-dir=/backups/incremental-backup-Monday  
  --incremental-base=/backups/initial-backup-1 --user=root  
  --password
```

```
Enter password:
```

```
mysqlbackup: INFO: MySQL server version is '5.5.23-log'.
```

```
mysqlbackup: INFO: Got some server configuration information from running  
server.
```

```
IMPORTANT: Please check that mysqlbackup run completes successfully.
```

```
At the end of a successful 'backup' run mysqlbackup  
prints "mysqlbackup completed OK!".
```

```
130826 19:02:43 mysqlbackup: INFO: MEB logfile created at  
/backups/incremental-backup-Monday/meta/MEB_2013-08-26.19-02-43_backup.log
```

```
-----  
Server Repository Options:  
-----
```

```
datadir = /usr/local/mysql/data/  
innodb_data_home_dir =  
innodb_data_file_path = ibdata1:10M:autoextend  
innodb_log_group_home_dir = /usr/local/mysql/data/  
innodb_log_files_in_group = 2  
innodb_log_file_size = 5242880  
innodb_page_size = Null  
innodb_checksum_algorithm = innodb
```



-----  
Backup Config Options:  
-----

```
datadir = /backups/incremental-backup-Monday/datadir
innodb_data_home_dir = /backups/incremental-backup-Monday/datadir
innodb_data_file_path = ibdata1:10M:autoextend
innodb_log_group_home_dir = /backups/incremental-backup-Monday/datadir
innodb_log_files_in_group = 2
innodb_log_file_size = 5242880
innodb_page_size = 16384
innodb_checksum_algorithm = innodb
```

mysqlbackup: INFO: Unique generated backup id for this is 13775581632965290

mysqlbackup: INFO: Creating 14 buffers each of size 16777216.

558

130826 19:02:45 mysqlbackup: INFO: Full Backup operation starts with following threads

1 read-threads    6 process-threads    1 write-threads

130826 19:02:45 mysqlbackup: INFO: System tablespace file format is Antelope.

130826 19:02:45 mysqlbackup: INFO: Starting to copy all innodb files...

130826 19:02:45 mysqlbackup: INFO: Found checkpoint at lsn 20135490.

130826 19:02:45 mysqlbackup: INFO: Starting log scan from lsn 20135424.

130826 19:02:45 mysqlbackup: INFO: Copying /usr/local/mysql/data/ibdata1 (Antelope file format).

130826 19:02:45 mysqlbackup: INFO: Copying log...

130826 19:02:45 mysqlbackup: INFO: Log copied, lsn 20135490.

130826 19:02:45 mysqlbackup: INFO: Completing the copy of innodb files.

130826 19:02:46 mysqlbackup: INFO: Preparing to lock tables: Connected to mysqld server.

130826 19:02:46 mysqlbackup: INFO: Starting to lock all the tables...

130826 19:02:46 mysqlbackup: INFO: All tables are locked and flushed to disk

130826 19:02:46 mysqlbackup: INFO: Opening backup source directory '/usr/local/mysql/data/'

130826 19:02:46 mysqlbackup: INFO: Starting to backup all non-innodb files in subdirectories of '/usr/local/mysql/data/'

130826 19:02:46 mysqlbackup: INFO: Copying the database directory 'menagerie'

130826 19:02:46 mysqlbackup: INFO: Copying the database directory 'my\_sensor\_network'

130826 19:02:46 mysqlbackup: INFO: Copying the database directory 'mysql'

130826 19:02:46 mysqlbackup: INFO: Copying the database directory 'newschema'

130826 19:02:46 mysqlbackup: INFO: Copying the database directory 'performance\_schema'

130826 19:02:46 mysqlbackup: INFO: Copying the database directory 'test\_arduino'

130826 19:02:46 mysqlbackup: INFO: Copying the database directory 'world'

130826 19:02:46 mysqlbackup: INFO: Copying the database directory 'world\_innodb'

130826 19:02:46 mysqlbackup: INFO: Completing the copy of all non-innodb files.

```

130826 19:02:47 mysqlbackup: INFO: A copied database page was
modified at 20135490.
    (This is the highest lsn found on page)
    Scanned log up to lsn 20137372.
    Was able to parse the log up to lsn 20137372.
    Maximum page number for a log record 724
130826 19:02:47 mysqlbackup: INFO: All tables unlocked
130826 19:02:47 mysqlbackup: INFO: All MySQL tables were locked for
0.954 seconds.
130826 19:02:47 mysqlbackup: INFO: Reading all global variables from the server.
130826 19:02:47 mysqlbackup: INFO: Completed reading of all global variables
from the server.
130826 19:02:47 mysqlbackup: INFO: Creating server config files
server-my.cnf and server-all.cnf in /backups/incremental-backup-Monday
130826 19:02:47 mysqlbackup: INFO: Full Backup operation completed successfully.
130826 19:02:47 mysqlbackup: INFO: Backup created in directory
'/backups/incremental-backup-Monday'
130826 19:02:47 mysqlbackup: INFO: MySQL binlog position: filename
mysql-bin.000026, position 71731

```

557

#### Parameters Summary

```

-----
Start LSN           : 20135424
End LSN             : 20137372
-----

```

mysqlbackup completed OK!



在 MEB 中，增量备份使用 InnoDB 的重做日志，应用从上次记录位置开始捕捉数据发生的变化。增量备份功能仅能用于 InnoDB 表。对于非 InnoDB 表，增量备份包括全量数据，而不仅仅是数据更改。

## 恢复数据

数据通过将前一节中所描述的日志进行恢复。这种操作基本上指向 MySQL 的实例的数据库的备份副本。它不将文件复制到正常的 MySQL 的 *datadir* 位置。如果想做到这一点，必须手动复制文件。这样做的原因是，*mysqlbackup* 实用程序拒绝覆盖任何数据。要启动一个 MySQL 实例，并使用你的数据备份，执行如下命令：

```

$ sudo ./mysqlbackup --defaults-file=/etc/my.cnf copy-back \
--datadir=/usr/local/mysql/data --backup-dir=/backups/initial-backup
MySQL Enterprise Backup version 3.9.0 [2013/08/23]
Copyright (c) 2003, 2013, Oracle and/or its affiliates. All Rights Reserved.

```

mysqlbackup: INFO: Starting with following command line ...

```
./mysqlbackup --defaults-file=/etc/my.cnf copy-back  
--datadir=/usr/local/mysql/data  
--backup-dir=/backups/initial-backup
```

IMPORTANT: Please check that mysqlbackup run completes successfully.

At the end of a successful 'copy-back' run mysqlbackup  
prints "mysqlbackup completed OK!".

130826 19:52:46 mysqlbackup: INFO: MEB logfile created at  
/backups/initial-backup/meta/MEB\_2013-08-26.19-52-46\_copy\_back.log

-----  
Server Repository Options:  
-----

```
datadir = /usr/local/mysql/data  
innodb_data_home_dir = /usr/local/mysql/data  
innodb_data_file_path = ibdata1:10M:autoextend  
innodb_log_group_home_dir = /usr/local/mysql/data  
innodb_log_files_in_group = 2
```

```
innodb_log_file_size = 5M  
innodb_page_size = Null  
innodb_checksum_algorithm = innodb
```

558

-----  
Backup Config Options:  
-----

```
datadir = /backups/initial-backup/datadir  
innodb_data_home_dir = /backups/initial-backup/datadir  
innodb_data_file_path = ibdata1:10M:autoextend  
innodb_log_group_home_dir = /backups/initial-backup/datadir  
innodb_log_files_in_group = 2  
innodb_log_file_size = 5242880  
innodb_page_size = 16384  
innodb_checksum_algorithm = innodb
```

mysqlbackup: INFO: Creating 14 buffers each of size 16777216.

130826 19:52:46 mysqlbackup: INFO: Copy-back operation starts with following  
threads 1 read-threads 1 write-threads

130826 19:52:46 mysqlbackup: INFO: Copying

```

/backups/initial-backup/datadir/ibdata1.
130826 19:52:46 mysqlbackup: INFO: Copying the database directory 'menagerie'
130826 19:52:46 mysqlbackup: INFO: Copying the database directory
'my_sensor_network'
130826 19:52:46 mysqlbackup: INFO: Copying the database directory 'mysql'
130826 19:52:46 mysqlbackup: INFO: Copying the database directory 'newschema'
130826 19:52:46 mysqlbackup: INFO: Copying the database directory
'performance_schema'
130826 19:52:46 mysqlbackup: INFO: Copying the database directory 'test_arduino'
130826 19:52:46 mysqlbackup: INFO: Copying the database directory 'world'
130826 19:52:46 mysqlbackup: INFO: Copying the database directory 'world_innodb'
130826 19:52:46 mysqlbackup: INFO: Completing the copy of all non-innodb files.
130826 19:52:46 mysqlbackup: INFO: Copying the log file 'ib_logfile0'
130826 19:52:46 mysqlbackup: INFO: Copying the log file 'ib_logfile1'
130826 19:52:47 mysqlbackup: INFO: Creating server config files server-my.cnf
and server-all.cnf in /usr/local/mysql/data
130826 19:52:47 mysqlbackup: INFO: Copy-back operation completed successfully.
130826 19:52:47 mysqlbackup: INFO: Finished copying backup files to
'/usr/local/mysql/data'

```

mysqlbackup completed OK!



你必须指定 `--defaults-file` 作为第一参数，紧跟在命令后面。如果不这么做，将会有下列错误信息显示：*mysqlbackup: unknown variable 'defaults-file=/etc/my.cnf'*。

## 559 离线备份

有件任务是必要的，就是为停止的服务器做离线备份。这个是 MEB 的一个亮点，MEB 能够进行所谓的“冷备份”：备份停止的服务器。执行全库备份时需要指定 `--no-connection` 和 `--defaults-file` 选项，同样可以指定 `--datadir`。下面举一个冷备份的例子（为了简洁忽略消息输出，这些消息和热备份是一样的）：

```

$ sudo ./mysqlbackup --defaults-file=/etc/my.cnf backup-and-apply-log \
--no-connection --datadir=/usr/local/mysql/data \
--backup-dir=/backups/emergency-cold-backup
MySQL Enterprise Backup version 3.9.0 [2013/08/23]
Copyright (c) 2003, 2013, Oracle and/or its affiliates. All Rights Reserved.

```

```

mysqlbackup: INFO: Starting with following command line ...
./mysqlbackup --defaults-file=/etc/my.cnf backup-and-apply-log

```



```
--no-connection --datadir=/usr/local/mysql/data
--backup-dir=/Users/cbell/source/emergency-cold-backup
```

IMPORTANT: Please check that mysqlbackup run completes successfully.

At the end of a successful 'backup-and-apply-log' run mysqlbackup prints "mysqlbackup completed OK!".

[...]

130826 20:09:09 mysqlbackup: INFO: Full backup prepared for recovery successfully.

mysqlbackup completed OK!

## 使用 MySQL 实用工具集进行数据库的导出和导入

MySQL 实用工具是 MySQL Workbench Tool 的一部分，我们将在第 17 章着重介绍相关细节，但是本小节介绍两个能备份和恢复数据的工具。当你要产生一份数据用来转换（批量或有针对性地更改数据）或阅读的时候可能会很有用。

第一个工具是 *mysqldbexport*。这个工具允许你读取数据库（数据库列表或者是所有数据库），同时可以输出成多种格式，包括 SQL 语句方式、逗号或 tab 隔离的列表，以及网格或垂直输出，就像 *mysql* 客户端的输出一样，可以重定向到文件以便于以后使用。

◀ 560

第二个工具是 *mysqldbimport*，这个工具读取 *mysqldbexport* 的输出。

两种工具都允许你导入仅对象定义、仅数据或两者都导入。这个有点像 *mysqldump*，在很多地方都很像，但是这些工具的选项更简洁（*mysqldump* 被抱怨需要很多选项）而且是用 Python 编写的，这样可以使得更专业的工程师能够根据自己的需要定制导出导入方式。

可以在第 17 章更深入了解 *mysqldbexport* 和 *mysqldbimport* 工具。

## mysqldump 工具

物理文件复制功能最常见的替代者就是 *mysqldump* 客户端应用。它是 MySQL 安装的一部分，最初是由 Igor Romanenko 捐献给 MySQL 的。*mysqldump* 创建了一组 SQL 语句集，重新运行这些 SQL 语句可以重建数据库。例如，运行备份时，输出结果包含了所有创建数据库及其表所需的 CREATE 语句，以及所有重建这些表数据的 INSERT 语句。

如果需要在数据文本上执行查找和替换操作，使用 *mysqldump* 可以很方便地完成该操作。只需备份数据库，使用文本编辑器编辑结果文件，然后恢复数据库以使更改生效。许多 MySQL 用户使用这种方法来更正所有由批量编辑而导致的错误。这比使用复杂的 WHERE 子句写 1000 个 UPDATE 语句容易多了。

使用 *mysqldump* 的缺点是：它比文件级（物理）备份的二进制文件复制（如 MEB、LVM 或简单的脱机文件复制）花费的时间要长，而且需要更大的存储空间。如果经常需要备份，并希望在系统出现故障后快速恢复，或者需要通过网络传输备份文件，那么这个时间成本将会非常大。

使用 *mysqldump* 可以备份所有数据库、特定的数据库集合，甚至还可以备份指定数据库中的某些表，如下所示：

```
mysqldump -uroot -all-databases
mysqldump -uroot db1, db2
mysqldump -uroot my_db tl
```

561

还可以使用 *mysqldump* 执行 InnoDB 表的热备份。--single-transaction 选项在备份开始的时候发出 BEGIN 语句，指示 InnoDB 存储引擎以一致性读的方式读取表。因此，你做的任何更改都会应用到表上，但是数据在备份时被冻结。然而，连接不能使用 DDL（数据定义语言）语句，如 ALTER TABLE、DROP TABLE、RENAME TABLE、TRUNCATE TABLE。这是因为一致性读无法隔离 DDL 更改。



--single-transaction 选项和 --lock-tables 选项是相互排斥的，因为 LOCK TABLES 采用的是隐式提交。

*mysqldump* 工具具有很多用于控制备份的选项，表 15-2 描述了一些比较重要的选项。完整的选项集合参见 MySQL 在线参考手册。

表15-2: 重要选项

选项	作用
--add-drop-database	在每个数据库前包含一个 DROP DATABASE 语句
--add-drop-table	在每个表之前包含一个 DROP TABLE 语句
--add-locks	在被包含的表的前面加上 LOCK TABLES，后面加上 UNLOCK TABLES
--all-databases	包含所有的数据库
--create-options	在 CREATE TABLE 语句中包含所有 MySQL 指定表的选项

选项	作用
--databases	只包含数据库的列表
--delete-master-logs	在 master 上, 在执行备份后删除二进制日志
--events	备份被包含的数据库中的事件
--extend-insert	另一种将所有的记录作为 VALUES 子句的 INSERT 语法
--flush-logs	在开始备份前刷新日志文件
--flush-privileges	在备份 <i>mysql</i> 数据库后包含一个 FLUSH PRIVILEGES 语句
--ignore-table=db.tbl	不备份指定的表
--lock-all-tables	在转储过程中锁定所有数据库中的所有表
--lock-tables	在包含表前锁定所有表
--log-error=filename	将警告和错误追加到指定文件中
--master-data[=value]	在输出结果中包含二进制日志文件名及其位置
--no-data	不写入任何表的行信息 (只包含 CREATE 语句)
--password[=password]	连接到服务器的密码
--port=port_num	连接时使用的 TCP/IP 端口号
--result-file=filename	结果输出到指定文件中
--routines	包括存储程序 (过程和函数)
--single-transaction	在从 slave 中导出数据之前执行 BEGIN SQL 语句, 这允许 InnoDB 表的一致性快照
--tables	重写 --databases 选项
--triggers	包含触发器
--where='condition'	只包含在 condition 中被选中的行
--xml	生成 XML 输出结果



还可以在 MySQL 配置文件中的 [mysqldump] 下使用这些选项。在大多数情况下, 只要删除行首的 -- 就可以指定选项。例如, 要产生 XML 输出, 需要在配置文件中包含 xml 选项。

*mysqldump* 的一个非常方便的功能是转储数据库模式。通常可以这样做: 使用一组 CREATE 命令 (不包括 INSERT 语句) 来重新创建所有的对象。这种用法有利于保存模式更改的历史记录。如果使用 --no-data 选项, 加上其他包含所有对象的选项 (如 --routines、--triggers), 可以使用 *mysqldump* 创建数据库模式。

请注意 --master-data 选项, 该选项有助于执行 PITR, 因为它保存了二进制日志信息, 与 InnoDB 热备份一样。

还有很多选项允许你控制该工具的运行。如果使用 SQL 语句创建备份是你的最佳选择,



那么可以研究一下其他选项，以更好地使用 *mysqldump*。

## 物理文件复制

最简单和最基础的 MySQL 备份的形式是简单文件复制。不幸的是，为了得到更好的结果，这种方法需要停止服务器。要执行文件复制，只需停止服务器，然后复制服务器上的数据目录和所有安装文件。一种常见的方法是使用 UNIX 中的 *tar* 命令创建归档，然后将这个归档文件移到另一个系统上，并恢复数据目录。

下面是一个典型的 *tar* 命令，备份一个数据库服务器上的数据，然后在另一个系统上恢复这些数据。在需要备份的服务器上执行以下命令，其中 *backup\_2013\_09\_09.tar.gz* 是要创建的文件，*/usr/loca/mysql/data* 是数据目录的路径：

```
tar -czf backup_2013_09_09.tar.gz /usr/loca/mysql/data/*
```

563 *backup\_2013\_09\_09.tar.gz* 文件所在的文件夹由两个服务器共享（或者必须手动将它复制到新的服务器上）。现在，在你需要恢复数据的服务器上，打开新安装的 MySQL 数据库的根目录，如果数据目录存在，删除现有的数据目录，然后执行以下命令：

```
tar -xvf ../backup_2013_09_09.tar.gz
```



前面讲过，最好使用有意义的文件名作为备份映像命名。

从这个例子可以看出，在操作系统级备份数据是很容易的事情。不仅得到了一个便于迁移的压缩归档文档，还可以从快速文本复制中获益。通过简单地复制数据目录中的单个文件或子目录，甚至可以执行选择性备份。

遗憾的是，*tar* 命令仅适用于 Linux 和 UNIX 平台。如果你在 Windows 平台上安装了 Cygwin，并且 Cygwin 包含了 *tar* 命令，那么在 Windows 平台上也可以使用 *tar* 命令。还可以在 windows 平台上使用 7zip 创建 *tar* 文件。

除了使用 Cygwin 和其他的 UNIX-on-Windows 包，Windows 中最接近 *tar* 命令的是资源管理器的文件夹归档功能或像 WinZip 这样的归档程序。打开 Windows 资源管理器，然后进入你的数据目录。不直接打开目录，而是右键单击数据目录，并从快捷菜单中选择压缩数据的选项，即选择 SendTo → Compressed (zipped) Folder，然后为 *.zip* 文件提供一个名称。



虽然物理文件复制是最快、最简单的备份形式，但是它要求关闭服务器。然而，如果能保证在文件复制过程中没有数据更新，就不需要锁定所有表和执行 `flush tables` 命令，然而并不推荐这样做，最好在复制文件之前将服务器离线（关闭）。



这与克隆 slave 的过程类似。具体如何使用文件复制克隆 slave 详见第 2 章。

此外，根据数据的大小而定，服务器不仅必须在复制文件时脱机运行，而且在以下情况下也必须脱机运行：任何额外数据加载时（如加载缓存项）、快速查找下使用内存表时等。因此，物理复制备份可能不适合某些安装项。

◀ 564

幸运的是，Tim Bunce 创建了一个名为 *mysqlhotcopy.sh* 的 Perl 脚本，可自动运行这个过程。它位于 MySQL 安装文件目录的 *./scripts* 文件夹下，它允许对数据库进行热备份。但是该脚本只能用来备份 MyISAM 或文档存储引擎，且只能在 UNIX 和 Netware 操作系统上运行。

*mysqlhotcopy.sh* 工具还有一些自定义功能，可以在 *mysqlhotcopy* 文档上找到更多关于它的信息（<http://bit.ly/mysqlhotcopy>）。

## 逻辑卷管理器快照

大多数 Linux 和某些 UNIX 系统提供了另一种强大的 MySQL 数据库的备份方法，该方法使用的技术称为逻辑卷管理器（Logical Volume Manager，LVM）。

Windows 中也有一个类似的技术叫卷影复制（Volume Shadow Copy）。遗憾的是，没有通用的工具用于对 LVM 中的任意分区或文件夹结构进行快照。不过，可以为整个驱动器进行快照，如果驱动器上只有数据库目录，那么该功能很有用。参看 Microsoft 的在线文档可了解更多信息。

LVM 是一个磁盘子系统，不需要使用陈旧的、复杂的和枯燥的磁盘工具，它可以帮助你简单快速地创建、删除并调整卷的大小。

备份的额外好处就是进行快照，即在不影响应用程序访问卷数据的情况下，复制活动卷。其思想是进行快照，快照的执行速度相对较快，然后备份该快照，而不是备份原始卷。LVM 中的快照技术使用这样的机制：跟踪快照开始后的一切更改，只存储更改的磁盘段。因此，快照比完全复制卷占用的空间少，而且当备份完成时，LVM 将复制快照时的所有文件。快照有效地冻结了数据。

使用 LVM 和快照备份数据库系统的另一个好处在于你如何使用卷。最好将 MySQL 安

装在一个单独的卷上，那样，所有的数据都在同一个卷上，就可以使用快照迅速创建备份。有些情况下可能会使用多个逻辑卷，比如一个表空间使用一个逻辑卷，甚至不同的 MyISAM 和 InnoDB 表使用不同的逻辑卷。

565 LVM 入门

如果你的 Linux 系统上没有安装 LVM，可以使用包管理器安装。例如，在 Ubuntu 上使用以下命令安装 LVM:

```
sudo apt-get install lvm2
```

虽然并非所有的 LVM 系统都相同，但是下面的 LVM 系统是基于标准 Debian 发行版开发的，而且在像 Ubuntu 这样的系统上运行良好。我们打算写一本关于 LVM 的完整教程，只是让你了解使用 LVM 进行数据库备份的复杂性。关于系统支持哪种 LVM 类型请参阅你的操作系统文档，或参看网上的指导文档。

在开始详细讨论 LVM 之前，首先花点时间了解一下 LVM 的基本概念。LVM 是分层实现的。最底层是磁盘，再上一层是分区，允许磁盘之间相互通信，再上面是物理卷，它是 LVM 的控制机制。可以向卷组中添加物理卷（卷组可以包含多个物理卷），一个卷组可以包含一个或多个逻辑卷。图 15-1 描述了文件系统、卷组、物理卷和块设备之间的关系。

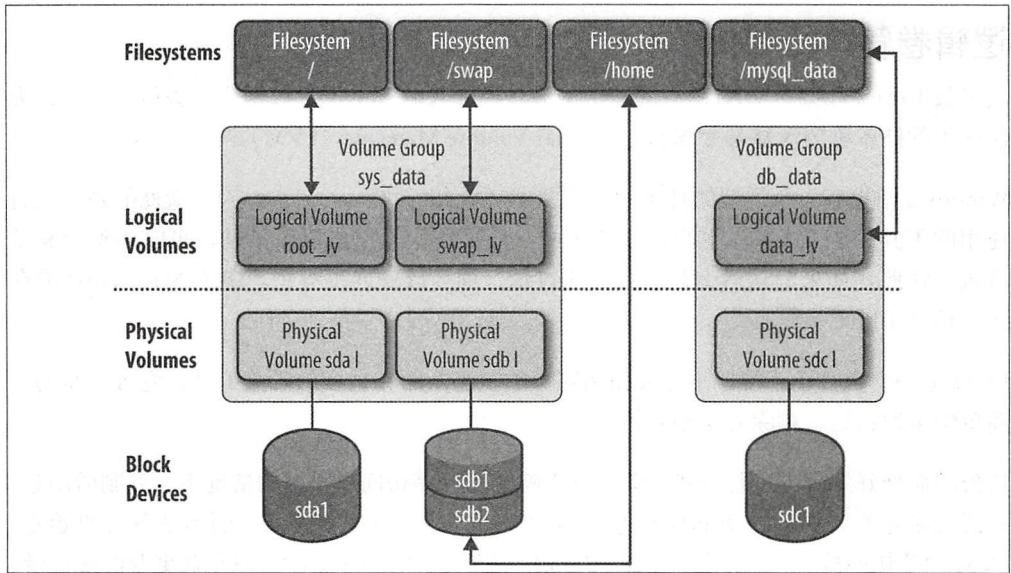


图15-1: LVM的分解图

566 逻辑卷可以作为一个正常挂载的文件系统，也可以作为一个快照逻辑卷。快照逻辑卷的创建是在备份中使用快照的关键。以下章节描述了如何开始使用 LVM 和备份数据。

有几个有用的命令是你应该熟悉的。下面列出了最常用的命令及其用途，更多信息请参考相关文档。

**pvcreate**

创建物理卷

**pvscan**

显示物理卷的详细信息

**vgcreate**

创建卷组

**vgscan**

显示卷组的详细信息

**lvcreate**

创建逻辑卷

**lvscan**

显示逻辑卷的详细信息

**lvremove**

删除逻辑卷

**mount**

挂载逻辑卷

**umount**

卸载逻辑卷

为了使用 LVM，需要有一个新磁盘，或者可被逻辑卸载的磁盘设备，操作过程如下（这个输出结果是在安装了 Ubuntu 9.04 的笔记本电脑上产生的）：

1. 创建一个现有的 MySQL 数据目录的备份：

```
tar -czf ~/my_backups/backup.tar.gz /dev/mysql/datadir
```

2. 为驱动器分区：

```
sudo parted  
select /dev/sdb  
mklabel msdos  
mkpart test  
quit
```

567 3. 为驱动器创建一个物理卷：

```
sudo pvcreate /dev/sdb
```

4. 创建卷组：

```
sudo vgcreate /dev/sdb mysql
```

5. 为数据创建逻辑卷，这里我们创建了一个 20GB 的卷：

```
sudo lvcreate -L20G -ndatadir mysql
```

6. 在逻辑卷上创建一个文件系统：

```
mke2fs /dev/mysql/datadir
```

7. 挂载逻辑卷：

```
sudo mkdir /mnt  
sudo mount /dev/mysql/datadir /mnt
```

8. 复制文档，并在你的逻辑卷上重建数据：

```
sudo cp ~/my_backups/backup.tar.gz  
sudo tar -xvf backup.tar.gz
```

9. 创建一个 MySQL 服务器实例，并使用 `--datadir` 指定逻辑卷上的文件夹：

```
./mysqld --console -uroot --datadir=/mnt
```



如果你想试用 LVM，建议使用一个允许丢失数据的磁盘。一个合适且廉价的选择是使用小型 USB 硬盘驱动。

以上介绍了逻辑卷的入门知识。在产品系统上开始使用 LVM 之前，请花一些时间去试用 LVM 工具，直到可以很熟练地运用它们。

## 使用 LVM 进行备份和还原

为了执行备份，需要刷新和临时锁定所有的表，紧接着执行快照，然后给所有表解锁。为了保证所有正在进行的事务执行结束，必须锁定表。下面给出了这个备份过程，以及相应操作的 shell 命令。



1. 在 MySQL 客户端执行 `FLUSH TABLES WITH READ LOCK` 命令。

2. 创建逻辑卷的快照（-s 选项表示快照）：

```
sudo lvcreate -L20M -s -n backup /dev/mysql/datadir
```

3. 在 MySQL 客户端执行 `UNLOCK TABLES` 命令（现在服务器可以恢复操作了）。

4. 装载快照：

```
sudo mkdir /mnts
sudo mount /dev/mysql/backup /mnts
```

5. 执行快照备份：

```
tar -[FIXTHIS]f snapshot.tar.gz /mnts
```

当然，快照的最佳用途是定期启动复制以进行下一次备份。网上有志愿者提供了自动执行上述过程的脚本，但是真正可靠的方式是删除快照然后重建，过程如下。

1. 卸载快照：

```
sudo umount /mnts
```

2. 删除快照（逻辑卷）：

```
sudo lvremove /dev/mysql/backup
```

然后重新创建快照并执行备份。如果有自己的脚本，建议在验证备份存档已经创建之后，添加一步删除快照，保证脚本执行了适当的清理操作。

如果需要恢复快照，只需恢复数据就可以了。LVM 真正的好处在于：使用 `tar` 工具创建快照和备份的所有操作，都可以通过创建定期运行的自定义脚本（如 `cron job`）实现，这个脚本可以帮助你进行自动备份。

## ZFS 中的 LVM

使用 Sun 公司的 ZFS 文件系统（Solaris 10 中可用）执行备份的流程和 Linux LVM 相似，这里描述一下它们的不同。

在 ZFS 中，在池（与卷组类似）中存储逻辑卷（Sun 将可读写的卷称为文件系统，而只读的称为快照）。创建备份或副本只需创建文件系统的快照。

使用下面的命令创建可管理的 ZFS 文件系统：

```
zpool create -f mypool c0d0s5
zfs create mypool/mydata
```

使用以下命令创建备份（执行新文件系统的快照）：

```
zfs snapshot mypool/mydata@backup_12_Dec_2013
```

使用以下命令将文件系统恢复到某个指定的备份：

```
cd /mypool/mydata
zfs rollback mypool/mydata@backup_12_Dec_2013
```

ZFS 不仅提供了完全卷（文件系统）备份，还支持指定文件恢复。

## XtraBackup

独立开源软件供应商 Percona 是 MySQL（实际上是 LAMP）领域的专业咨询公司，它创建了一个名为 XtraDB 的存储引擎，该存储引擎是基于 InnoDB 存储引擎的开源存储引擎。XtraDB 改进了 InnoDB，具备更好的硬件伸缩能力，且向后兼容 InnoDB。

Percona 创建了 XtraDB 的热备份方案——XtraBackup。这个工具优化后可供 InnoDB 和 XtraDB 使用，也可以用来备份和还原 MyISAM 表。它提供了很多备份相关的特性，包括压缩和增量备份。

可以从 Launchpad 上下载源代码构建 XtraBackup，网址为 <https://launchpad.net/percona-xtrabackup>。在大多数平台上都可以编译和执行 XtraBackup，它与 MySQL 5.0，5.1，5.5 和 5.6 版本兼容。

## 备份方法的比较

MySQL Enterprise Backup、MySQL 实用工具、*mysqldump* 及其他第三方备份方法的区别体现在几个重要的方面，还有很多细微的差别。这里比较了各种备份方法，主要包括以下几个方面：是否允许热备份、成本、备份速度、恢复速度、备份类型（逻辑或物理）、平台限制（操作系统）和支持的存储引擎。表 15-3 列举了本章提到的各种备份方法及其比较项。

表 15-3: 备份方法的对比

	MySQL 企业版备份	MySQL 实用工具	mysqldump	物理 复制	LVM、 ZFS 快照	XtraBackup
热备份?	是 (仅 InnoDB)	是 (仅 InnoDB)	是 (仅 InnoDB, 需要添加 --single- transaction 选项)	否	是 (需要 全局表刷 新锁)	是 (仅 InnoDB 和 XtraDB)
开销	免费	免费	免费	免费	免费	免费
备份速度	中等	慢	慢	快	快	中等
恢复速度	快	慢	慢	快	快	快
类型	物理	逻辑	逻辑	物理	物理	物理
OS	所有	所有	所有	所有	支持 LVM 的操作系统	所有
引擎	所有	所有	所有	所有	所有	InnoDB、 XtraDB、 MyISAM

570

表 15-3 有助于规划数据恢复流程，帮助你找到满足需求的最佳工具。例如，如果需要在 InnoDB 数据库上进行热备份，并且成本不是问题，那么 MySQL 企业版备份将是你的最佳选择。另外，如果速度是主要考虑因素，要求备份所有数据库（所有存储引擎），并且操作系统为 Linux，那么 LVM 是一个很好的选择。

## 备份和 MySQL 复制

使用 MySQL 复制进行备份有两种方法。在前面的章节已经介绍了 MySQL 复制及其在横向扩展和高可用性上的用处。本章介绍 MySQL 复制与备份有关的两个常见用途，包括使用复制创建数据备份副本，以及使用前面创建的备份进行 PITR（即时恢复）。

### 备份和恢复

使用额外的服务器创建备份是很常见的，这样可以在不影响 master 的情况下创建备份，因为你可以使备份服务器离线，然后执行任何操作。

### PITR

即使定期创建备份，仍可能需要将服务器恢复到某个精确的时间点。通过合理地管理备份，确实可以将服务器恢复到指定的秒时间点。这对于恢复人为错误（例如敲错命令或输入不正确的数据），或者还原不必要的更改非常有用。各种情况都可能发生，它们都需要有正确的备份。

## 使用复制进行备份和恢复

一般来说，备份的缺陷在于只能在特定的时间创建（为了不影响其他操作，经常在夜间执行）。如果有个问题需要将 master 恢复到备份创建后的某个时间点，你就惨了吗？不是的！如果将备份与二进制日志结合使用，那么解决这个问题不仅仅是可能的，而且很容易。

二进制日志记录了在数据库运行时对该数据库所做的所有更改，所以，通过恢复正确的备份并将二进制日志回退到合适的时间点，就可以将服务器恢复到一个精确的时间点上。

当然，恢复过程中最重要的一步是恢复。所以在概述执行备份的流程之前，让我们先讨论如何执行恢复。

### PITR

复制中的备份最常见的用途就是 PITR，即通过将系统恢复到最近正确的状态来使系统从错误（如数据丢失或硬件故障）中恢复，从而减少数据丢失。这么做的前提是至少进行了一次备份。

一旦修复了服务器，就可以还原最新备份映像，然后以二进制日志名及其位置为起点应用二进制日志。

下面描述了使用备份系统执行 PITR 的过程：

1. 将服务器还原到事件发生后的一个操作状态。
2. 找到你需要还原的数据库的最新备份。
3. 还原最新备份映像。
4. 使用 *mysqlbinlog* 工具应用二进制日志，将最新备份的开始位置（或开始时间）作为二进制日志的开始执行点。

这时你可能有个疑问：“备份之后应该使用哪个二进制日志来执行 PITR 呢？”这取决于备份的方式。如果在运行备份前刷新了二进制日志，那么需要使用当前日志（最近打开的日志文件）的名称和位置。如果在运行备份前没有刷新二进制日志，则使用前一个日志的名称和位置。



更简单的情况是，PITR 在备份前总会刷新日志。这样，开始点就是文件的开头。



## 在错误被复制后还原

现在让我们看看备份如何帮助你恢复复制拓扑中的无意更改。假设某个用户做了一个灾难性(但却有效的)更改,并且这个更改被复制到了所有的 slave 上。这时复制帮不上忙,但是备份系统可以拯救你。

下面的步骤用来恢复复制拓扑中数据的无意更改:

1. 删除 master 上的数据库。
2. 停止复制。
3. 还原 master 上的事件发生之前的最新备份。
4. 记录 master 上当前二进制日志的位置。
5. 还原 slave 上的事件发生之前的最新备份。
6. 在 master 上执行 PITR, 执行流程如上一节所示。
7. 从记录位置重启复制, 并运行 slave 同步。

总之, 一个好的备份策略不仅仅是防止数据丢失的必要保护手段, 同时也是一个重要的复制工具。

## 恢复示例

现在让我们看一个具体示例。假设你每天凌晨 2 点定期创建备份, 并将备份映像存储在某个地方备用。在这个例子中, 假设所有的二进制日志都是可用的, 一个都没有删除。实际上会定期清除二进制日志, 以减少磁盘的占用空间。后面我们再考虑如何处理这种情况。

你已经将数据库恢复到 2013-12-19 12:54:23 时的状态, 因为这个时刻热心的助手不经意间将经理最心爱的照片删除了, 她在清理桌子的同时也清理了经理的电脑:

1. 找到 2013-12-19 12:54:23 之前的备份映像。

实际上, 选择哪个备份映像并没有什么区别, 但是为了节省恢复的时间, 应该选择最近的备份, 即当天上午的备份映像。

2. 还原备份映像, 创建一个数据库在 2013-12-19 02:00:00 时刻的精确副本。
3. 找出 2013-12-19 02:00:00 到 2013-12-19 12:54:23 之间的所有二进制日志文件。开始时间之前和结束时间之后的事件都不重要, 但是这些日志文件必须涵盖这个时间段内的所有事件。

4. 使用 `mysqlbinlog` 工具回放二进制日志文件，指定开始时间为 2013-12-19 02:00:00 和结束时间为 2013-12-19 12:54:23。

现在可以告诉经理他最心爱的照片回来了。

要想自动执行这个操作，必须做一些记录，这些记录将告诉你备份时需要保存什么。下面了解一下恢复时需要哪些信息。

- 为备份映像标记开始时间和结束时间，可以正确使用它们。这一点很重要，有助于选择使用哪个备份。
- 还需要备份二进制日志的位置。这个信息是必需的，因为回放二进制日志文件的开始时间不够精确。
- 还需要了解每个二进制日志文件记录信息的时间范围。严格地说，这不是必需的，但是很有帮助，因为它可以使你不必处理恢复映像的所有二进制日志文件。但是 MySQL 服务器不是自动执行的，你需要自己处理。
- 你不能永远保留这些文件，所以需要给所有信息、备份映像和二进制日志排序，这样当需要释放一些磁盘空间时，可以很容易地将它们归档。

## 恢复映像

为了帮助你管理可管理块中的备份信息，可引入恢复映像的概念。恢复映像只是一个虚拟容器而不是一个物理实体：它只包含执行恢复必需的所有信息。

图 15-2 显示了一个恢复映像序列及每个恢复映像所包含的内容。序列中最后一个恢复映像比较特殊，称为开放恢复映像（open recovery image）。该恢复映像仍然处于添加更改的状态，因此它没有结束时间。其他的恢复映像称为封闭恢复映像（closed recovery image），且有结束时间。

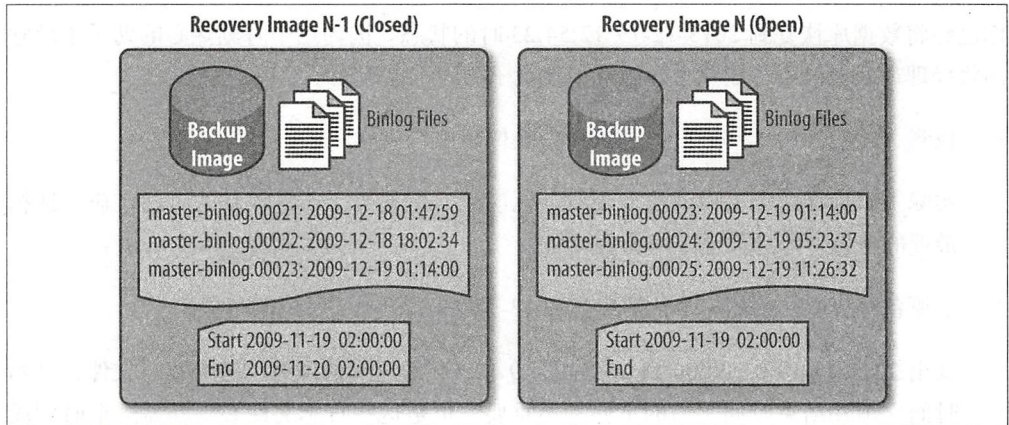


图15-2: 恢复映像序列和内容

每个恢复映像都包含了执行恢复必需的信息：

#### 备份映像

备份映像在还原数据库时需要。

#### 二进制日志文件集合

二进制日志文件必须覆盖恢复映像的整个时间范围。

#### 开始时间和可选的结束时间

这些是恢复映像代表的开始时间和结束时间。开放恢复映像没有结束时间，这样不便于归档，所以我们认为它的结束时间就是当前时间。



二进制日志文件通常包含开始时间和结束时间范围之外的事件，但是恢复映像应该包含该时间范围内的所有事件。

#### 每个二进制日志文件的名称和开始时间列表

提取二进制日志文件需要每个文件的名称、开始时间和结束时间。可以使用 `mysqlbinlog` 提取二进制日志文件的开始时间。

### 备份过程

备份程序收集归档和恢复所需的所有信息和结构。假设已经有一个  $\text{Image}_1$  到  $\text{Image}_{n-1}$  的恢复映像序列，现在要创建第  $n$  个恢复映像。

1. 使用你最喜欢的方法创建备份，并记录备份映像的名称及其所代表的二进制日志位置。该二进制日志位置还包含二进制日志文件名。

如果使用脱机备份工具，则备份时间就是锁定表的时间，锁定数据库后使用 `SHOW MASTER STATUS` 命令得到二进制日志位置。

2. 使用以下参数创建一个新的开放恢复映像  $\text{Image}_n$ 。

- `Backup(Imagen)` 是第一步中的备份映像。
- `Position(Imagen)` 是第一步中的二进制日志位置。
- `BinlogFiles(Imagen)` 未知，但第一个文件是第一步中的二进制日志文件名。
- `StartTime(Imagen)` 是映像的开始时间，来自于第一步中的二进制日志位置的事件。

3. 关闭  $\text{Image}_{n-1}$ ，请注意以下几点：

- $\text{BinlogFiles}(\text{Image}_{n-1})$  现在是从  $\text{Position}(\text{Image}_{n-1})$  到  $\text{Position}(\text{Image}_n)$ 。
- $\text{EndTime}(\text{Image}_{n-1})$  现在与  $\text{StartTime}(\text{Image}_n)$  相同。

## 使用 Python 完成 PITR

为了统一管理各种备份方法，我们已经在示例 15-1 中创建了 `PhysicalBackup` 类，这个类包含两个方法：

```
class PhysicalBackup.PhysicalBackup(image_name)
```

类的构造器，传入备份或恢复服务器所需的映像名称。

```
PhysicalBackup.backup_from(server)
```

该方法将创建服务器的备份，并存储该备份，该备份名称为构造器中传入的映像名。

```
PhysicalBackup.restore_on(server)
```

该方法将使用备份映像在服务器上执行备份和还原。

576 用类来表示备份方法，这样易于更换其他备份方法。

示例15-1：表示物理备份方法的类

```
class BackupImage(object):
```

```
    "Class for representing a backup image"
```

```
    def __init__(self, backup_url):
```

```
        self.url = urlparse.urlparse(backup_url)
```

```
    def backup_server(self, server, db):
```

```
        "Backup databases from a server and add them to the backup image."
```

```
        pass
```

```
    def restore_server(self, server):
```

```
        "Restore the databases in an image on the server"
```

```
        pass
```

```
class PhysicalBackup(BackupImage):
```

```
    "A physical backup of a database"
```

```
    def backup_server(self, server, db="*"):
```

```
        datadir = server.fetch_config().get('datadir')
```

```
        if db == "*":
```

```
            db = [d for d in os.listdir(datadir)
```

```
                    if os.path.isdir(os.path.join(datadir, d))]
```

```
        server.sql("FLUSH TABLES WITH READ LOCK")
```



```

position = replicant.fetch_master_pos(server)
if server.host != "localhost":
    path = basename(self.url.path)
else:
    path = self.url.path
server.ssh(["tar", "zpscf", path, "-C", datadir] + db)
if server.host != "localhost":
    subprocess.call(["scp", server.host + ":" + path, self.url.path])
server.sql("UNLOCK TABLES")
return position

def restore_server(self, server):
    if server.host == "localhost":
        path = self.url.path
    else:
        path = basename(self.url.path)

    datadir = server.fetch_config().get('datadir')

    try:
        server.stop()
        if server.host != "localhost":
            call(["scp", self.url.path, server.host + ":" + path])
        server.ssh(["tar", "zxf", path, "-C", datadir])
    finally:
        server.start()

```

577

接下来介绍恢复映像的表示，如示例 15-2 所示。该恢复映像存储了 5 个字段：

**RecoveryImage.backup\_image**

使用的备份映像。

**RecoveryImage.start\_time**

恢复映像的开始时间。

**RecoveryImage.start\_position**

备份映像表示的二进制日志位置。用该值代替开始时间作为回放二进制日志文件的开始点，这样比较准确。使用开始时间会出错，因为一秒内可能执行大量事务，它会选择标记为该时间的第一个事件，而真正的开始位置可能在其他地方。

**RecoveryImage.binlog\_files**

恢复映像的二进制日志文件列表。

`RecoveryImage.binlog_datetime`

将二进制日志文件名映射到日期/时间的字典，即二进制日志文件中的第一个事件的日期/时间。

另外，恢复映像还必须包含以下方法：

`RecoveryImage.contains(datetime)`

判断恢复映像是否包含 `datetime`。由于二进制日志文件可能在半秒的时刻轮换，所以必须含有 `end_time`。

`RecoveryImage.backup_from(server)`

通过创建 `server` 的备份来创建一个新的开放恢复映像，并收集备份的相关信息。

`RecoveryImage.restore_to(server, datetime)`

还原服务器上的恢复映像，应用直到 `datetime` 之前（包括 `datetime`）发生的所有更改。这里假设 `datetime` 在恢复映像的时间范围内。如果 `datetime` 在恢复映像的开始时间之前，则不应用任何更改；如果 `datetime` 在恢复映像的结束时间之后，则应用所有的更改。

示例15-2：一个恢复映像的实例

```
class RecoveryImage(object):
```

```
    def __init__(self, backup_method):
```

```
        self.backup_method = backup_method
```

```
        self.backup_position = None
```

```
        self.start_time = None
```

```
        self.end_time = None
```

```
        self.binlog_files = []
```

```
        self.binlog_datetime = {}
```

```
    def backup_from(self, server, datetime):
```

```
        self.backup_position = backup_method.backup_from(server)
```

```
    def restore_to(self, server):
```

```
        backup_method.restore_on(server)
```

```
    def contains(self, datetime):
```

```
        if self.end_time:
```

```
            return self.start_time <= datetime < self.end_time
```

```
        else:
```

```
            return self.start_time <= datetime
```

因为管理恢复映像需要处理多个映像，所以在示例 15-3 中引入了 `RecoveryImageManager` 类。

除了构造方法外它还有两个方法：

`RecoveryImageManager.point_in_time_recovery(server, datetime)`

在 `server` 上执行 PITR，恢复到 `datetime` 时刻。

`RecoveryImageManager.point_in_time_backup(server)`

为 PITR 创建 `server` 的备份。

恢复映像管理器跟踪所有的恢复映像及使用的备份方法。这里假定所有的恢复映像使用相同的备份方法，但这不是必需的。

示例 15-3：RecoveryImageManager 类

```
class RecoveryImageManager(object):
    def __init__(self, backup_method):
        self.__images = []
        self.__backup_method = backup_method

    def point_in_time_recovery(server, datetime):
        from itertools import takewhile
        from subprocess import Popen, PIPE

        for im in images:
            if im.contains(datetime):
                image = im
                break
        image.restore_on(server)

    def before(file):
        return image.binlog_datetime(file) < datetime

    files = takewhile(before, image.binlog_files)
    command = ["mysqlbinlog",
               "--start-position=%s" % (image.backup_position.pos),
               "--stop-datetime=%s" % (datetime)]
    mysqlbinlog_proc = Popen(mysqlbinlog_command + files, stdout=PIPE)

    mysql_command = ["mysql",
                    "--host=%s" % (server.host),
                    "--user=%s" % (server.sql_user.name),
                    "--password=%s" % (server.sql_user.password)]
    mysql_proc = Popen(mysql_command, stdin=mysqlbinlog_proc.stdout)
    output = mysql_proc.communicate()[0]
```

◀ 579

```
def point_in_time_backup(self, server):
    new_image = RecoveryImage(self.__backup_method)
    new_image.backup_position = image.backup_from(server)
    new_image.start_time = event_datetime(new_image.backup_position)

    prev_image = self.__images[-1].binlog_files
    prev_image.binlog_files = binlog_range(prev_image.backup_position.file,
                                           new_image.backup_position.file)
    prev_image.end_time = new_image.start_time

    self.__images.append(new_image)
```

## 自动备份

自动备份是相当容易的。上一节演示了如何使用复制进行备份和恢复，本节将介绍备份和恢复的非复制方法的一般流程。

你可能会遇到的唯一问题是提供自动给备份映像文件命名的机制。解决办法有很多，示例 15-4 给出了使用备份时间给文件命名的方法。将这个备份方法添加到 Python 库中以完善你的复制方法。这个库与前面章节中的库一样。用你的备份解决方案替换可执行命令 `[BACKUP COMMAND]`。

示例15-4： 备份脚本

```
#!/usr/bin/python
```

```
import MySQLdb, optparse

# --
# Parse arguments and read Configuration
# --
parser = optparse.OptionParser()
parser.add_option("-u", "--user", dest="user",
                  help="User to connect to server with")
parser.add_option("-p", "--password", dest="password",
                  help="Password to use when connecting to server")
parser.add_option("-d", "--database", dest="database",
                  help="Database to connect to")
(opts, args) = parser.parse_args()
```



```

if not opts.password or not opts.user or not opts.database:
    parser.error("You have to supply user, password, and database")

try:
    print "Connecting to server..."
    #
    # Connect to server
    #
    dbh = MySQLdb.connect(host="localhost", port=3306,
                           unix_socket="/tmp/mysql.sock",
                           user=opts.user, passwd=opts.password,
                           db=opts.database)

    #
    # Perform the restore
    #
    from datetime import datetime

    filename = datetime.time().strftime("backup_%Y-%m-%d_%H-%M-%S.bak")
    dbh.cursor().execute("[BACKUP COMMAND]%s" % filename)
    print "\nBACKUP complete."

except MySQLdb.Error, (n, e):
    print 'CRITICAL: Connect failed with reason:', e

```

如果不需要创建备份映像的名字，那么自动还原就更简单了。然而，依据你的安装、使用 and 配置，可能需要添加一些命令以保证与其他应用或用户之间的交互不受破坏。示例 15-5 是一个可以加入 Python 库的常用还原方法，用来完善你的复制方法。用你的备份解决方案替换可执行命令 `[RESTORE COMMAND]`。

示例 15-5: 恢复脚本

```

#!/usr/bin/python

import MySQLdb, optparse

# --
# Parse arguments and read Configuration
# --
parser = optparse.OptionParser()
parser.add_option("-u", "--user", dest="user",
                  help="User to connect to server with")

```

```

parser.add_option("-p", "--password", dest="password",
                  help="Password to use when connecting to server")
parser.add_option("-d", "--database", dest="database",
                  help="Database to connect to")
(opts, args) = parser.parse_args()
if not opts.password or not opts.user or not opts.database:
    parser.error("You have to supply user, password, and database")

try:
    print "Connecting to server..."
    #
    # Connect to server
    #
    dbh = MySQLdb.connect(host="localhost", port=3306,
                          unix_socket="/tmp/mysql.sock",
                          user=opts.user, passwd=opts.password,
                          db=opts.database)

    #
    # Perform the restore
    #
    from datetime import datetime

    filename = datetime.time().strftime("backup_%Y-%m-%d_%H-%M-%S.bak")
    dbh.cursor().execute("[RESTORE COMMAND]%S",
                          (database, filename))
    print "\nRestore complete."
except MySQLdb.Error, (n, e):
    print 'CRITICAL: Connect failed with reason:', e

```

从示例 15-5 可以看出，可以自动执行还原。然而，大多数人喜欢手动执行还原。如果在测试环境下从基线开始恢复，并且需要将数据库还原到某个可知状态，这时自动还原可能比较有用。另外，自动还原还可以用于开发系统中为项目保留特定的环境。

## 小结

本章学习了 IA，并研究了它对 IT 专业人士最有影响的几个方面。了解了灾难恢复规划的重要性，如何制订自己的灾难恢复计划，以及数据库系统为什么是灾难恢复不可分割的一部分。最后，研究了通过定期备份来保护 MySQL 数据的几种方法。

接下来的几章将讨论更高级的 MySQL 话题，包括 MySQL 企业版、云计算和 MySQL 集群。

Joel 看了一眼终端窗口，又敲了个命令查看他的数据库备份。他建立了一个循环备份脚本来备份所有的数据库，相信这个简单的操作没什么问题。他叹息道，这仅仅只是他的恢复计划的开始，希望多写些脚本进行试验，然后安排他的第一个灾难计划会议。他已经想好了几个小组成员。突然，一阵敲门声打断了他的思路。

“订购媒体了吗,Joel? 那个计划怎么样了?”Summerson 先生问道。Joel 笑了笑说：“是的，我确定我可以备份整个数据库了，使用 .....”

“好的，很好，Joel。我不需要知道细节信息，只要远离审计员就好了，是吧？”

Joel 笑着点点头，然后老板就去询问另一个员工的工作了。Joel 怀疑他的老板是否真的了解要达到目标需要多大的工作量。他打开邮箱开始写邮件，请求分配更多的人手和资源。

# MySQL企业版监控

Joel 打开另一个终端窗口观察输出。他揉揉眼睛放松了下视神经。当他试图监控分散在三个地点的所有服务器时，屏幕上顿时变得乱七八糟，过了很久他才拿到需要的报告。

当只有几台服务器的时候，他曾试图使用电子表格来存储数据。但现在已经超过 30 台服务器，工作也随之枯燥起来。他的朋友曾试图说服他购买企业级的监控工具套件，但老板认为这样无法立即给公司带来收入，所以拒绝了该项议案。

“嗨，Joel。”

Joel 抬起头，看到他在客户支持部门的朋友 Doug 斜靠在他的门边，手里还拿着杯咖啡。“嗨，” Joel 说。

“你看上去需要休息一下。”

“没时间啊。我需要写一份所有服务器的状态报告，但是不检查完所有服务器，我写不出来。”

“哇，这样干真费劲。”

“嗯，我知道有些企业套件能很容易地解决这些问题，但即使 Summerson 先生批准，我也不知道该买哪一种。”

“嗯，如果你能向他证明这一切需要多少时间……” Doug 一边说，一边剧烈地摇着他的咖啡杯。

Joel 想了想，说，“要是我能把埋头蛮干跟使用好工具之间的区别展示给他看就好了……”



“好主意。来杯咖啡吧？我请客。”

Joel 跟着 Doug 一起来到休息间，聊了聊他们最近参加过的 Dave Matthews 乐队的音乐会。

Joel 回到办公室以后，就开始了解 Oracle 的 MySQL 企业版产品。

584

监控一组服务器的工作量很大。有很多可以妥善监管服务器的工具，虽然只要你了解了它们是做什么的，就能很容易地用起来，但是仍需要做点工作才能运行它们。脚本和一些有创意的 Web 应用有助于管理各种工具的运行及数据的收集，但是随着被监控服务器的数量的增加，搜集和分析所有数据将变得十分麻烦。

到目前为止，在本书中讨论过的监控技术还不能管理多台服务器。事实上，在一个拥有很多服务器的组织机构中，都是由多个全职技术人员进行手动监控和报告的。

本章主要讨论监控，并涉及部分管理。一般来说，如果使用管理工具自动执行命令，例如运行脚本来填充表、执行表维护等，可以大大减轻管理员的负担。

幸运的是，这个问题不是才出现的，也不是不能解决的。事实上，有几种企业监控套件可以很容易地监控大量服务器。

其中最容易被忽略的监控 MySQL 的工具是 MySQL 企业监控器 (MEM)，它已经附带在 MySQL 企业版中。MEM 工具可以大大改善监控和预防性维护，并大大减少诊断时间和停机时间。虽然这是一个付费工具，但是一个维护良好的数据中心节约下来的钱要远远超过购买该工具的开销。

本章介绍 MySQL 企业 3.0 版套件的工具，并展示这些工具是如何在保持 MySQL 服务器的性能和高可用性的情况下，有效地节约时间的，还将介绍在复杂复制拓扑结构上运行监控工具的实例。

## MySQL 企业版监控入门

MEM 是 MySQL 企业版中的一个组件。当购买了 MySQL 企业版的商业版许可，就可以获取 MySQL 企业版的所有工具。

MySQL 企业版是在 2006 年发布的，包括企业 MySQL 服务器发行版、一套监控工具及产品支持服务。这个新包是为使用 MySQL 管理数据的客户设计的。在早期，MySQL 认识到组织机构对稳定性和可靠性的需求，MySQL 企业版刚好满足这个需求。

585



如果你尚不准备购买 MySQL 企业版，或者想试用一段时间后再考虑买不买，可以在 Oracle 软件发行云上下载这个软件的试用版本。如果你是 Oracle 客户，那么也可以在 Oracle 技术支持网站上找到可下载的 MySQL 产品。

下面描述了 MySQL 企业版订阅服务的可用选项，并概述了其安装流程。后面的章节更详细地描述了它的特性和益处。

## 产品

MySQL 企业版套件中有 4 种收费版本，最高级的版本拥有 MySQL 企业版的所有功能，其中包含一系列用于帮助企业管理大型 MySQL 数据库的工具和产品。你也可以购买 MySQL 服务器的商业许可证，即 MySQL 标准版，它提供了基本的 MySQL 服务和专业技术支持。为了特殊需求，Oracle 提供了两套方案，MySQL 集群版提供给需要在内存数据库中高速处理大量冗余数据的企业，MySQL 经典版是为 ISV/OEM 厂商配置的。

有这么多版本可供选择，你可以随着业务的增加再添加不同的功能。如果需要了解关于 MySQL 企业版的更多信息，或者想获得专业的技术支持（如了解 MEM 高级工具），可以联系 Oracle MySQL 销售团队。登录 MySQL 产品服务网站可以获取销售团队的电话或邮箱。

## 剖析 MySQL 企业监控器

MySQL 企业监控器包含一个托管在其 Web 服务器(Tomcat)上的基于 Web 的 MEM 应用，以及一个被称为代理(agent)的独立的 MySQL 服务器实例，这个实例用于存放安装在 MySQL 服务器上的其他应用收集的指标。MEM 将这些指标组合成报告，这将可以帮助你实施基于 MySQL 研究和专业知识的最佳实践。这个报告展示在网页面板上。在本章后面我们将讨论如何安装和使用这些组件。

因此，MEM 由两个组件组成。Web 服务器和被称为 MEM 服务管理（有时候被称为 MySQL 企业监控或简单监控）的 MySQL 服务器实例。安装在 MySQL 服务器上的用于监控 MySQL 实例和收集数据的组件被称为 MEM 代理（或者简单的 MySQLAgent）。虽然 MEM 的有些部分不是特别需要一个或多个 MySQL 服务器实例，但是这是 MEM 运行的先决条件。这些组件和流程的信息图显示在图 16-1 中。我们将在后面的章节中讨论这些组件的详情。

现在已经了解了 MEM 的各个组件，接下来让我们看看如何安装这些组件并让 MEM 运行起来。

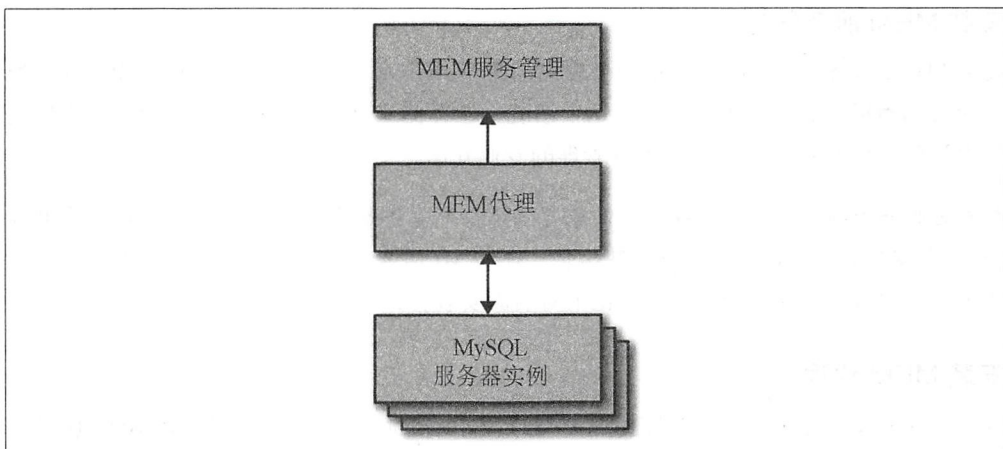


图16-1: MEM的剖析

## 安装概述

本书不提供安装说明的完整信息，不过我们将简要描述安装和配置 MySQL 企业套件的必要步骤。幸运的是，这个安装过程非常容易。

例如，如果需要进行一个新的安装，安装步骤如下：

1. 下载、安装和启动服务管理器。安装 MEM 服务管理器组件，其中包括指标库（独立于 MySQL 之外安装的），不同操作系统需要安装不同的指标库。例如，需要在 Mac OS X 系统上安装名为 *mysqlmonitor-3.0.0.2887-osx-installer.app* 的组件。
2. 使用浏览器打开服务管理器（开启自动安装），并使用 first-time 安装界面配置你的系统。还可以通过 <http://localhost:18443> 访问安装配置面板（Dashboard）。
3. 在 MySQL 服务器上安装监控代理，不同操作系统上需要安装不同版本的代理。例如，在 Linux 系统中，需要安装 *mysqlmonitoragent-3.0.0.2887-linux-glibc2.3-x86-64bit-installer.bin*。
4. 在配置面板中配置生产环境。你可以给服务器分组，并给每个服务器设置详细的名称。



可以从在线 MEM 参考手册中找到 MEM 安装的其他信息。

587



## 安装 MEM 服务管理器

安装 MEM 服务管理器组件包括一个自包含的 Web 服务器和安装在系统中的用于存储配置面板和指标集的 MySQL 实例。这个系统用于存储来自每个监控代理的数据。安装过程很简单，且不需要了解任何 Web 管理的专业知识。

在安装服务管理器组件的过程中，你需要确定每个服务器的账号名称和地址（例如 IP 地址），在安装监控代理时会用到这些信息。在线 MEM 参考手册的入门篇中介绍了所有这些选项。你可以打印这份指南，并记下对后面安装有帮助的信息。

## 安装 MEM 代理

监控代理的安装也很简单。当监控服务器启用并运行后，就可以在你的 MySQL 服务实例所在的每台服务器上安装一个代理。有些系统可能需要手动启动代理，用户指南中介绍了这样做的原因。

可以将监控代理和监控 MySQL 实例安装在不同的服务器上，最好是将它们安装在相同的服务器上。这样的话，代理就可以将操作系统的统计信息、性能数据和配置参数发送给配置面板。如果你在其他系统上安装代理，那么配置面板上将不会显示任何 MySQL 的相关报告。

可以使用现有的 MySQL 服务器，但是为了获得最大的投资回报率，最好使用你购买的 MySQL 企业版本许可证提供的版本。一旦你的数据库服务器配置和运行稳定后，就可以安装 MySQL 企业监控和监控代理。

首先需要将 MEM 服务安装在与所有需要监控的服务器相连接的机器上（我们经常推荐使用 MEM 监控你的 MySQL 服务器）。在安装过程中，需要记下这个服务器的主机名称或 IP 地址，以及代理访问所需的用户名和密码。安装过程很简单，在 MySQL 企业版网站上有详细介绍。



MEM 安装时有许多用户账号：MEM 管理员账号，代理访问 MEM 服务器的账号，和两个代理访问运行在每个 MySQL 服务器上的监控代理的账户。如果把这些账号搞混淆了，将会导致安装失败。

一旦 MEM 服务启动并运行在监控服务器上，就可以开始在每个 MySQL 服务器上安装监控代理。当在安装监控代理时，需要提供监控代理访问 MySQL 服务器的用户名和密码。每个服务器最好使用相同的访问账号。与安装 MEM 服务相同，安装监控代理也很简单。只需要记录所有使用过的用户账号名和密码。需要在安装监控代理的过程中使用与安装 MEM 相同的账号和密码。安装完成后，服务器应该在几分钟内显示在 MEM 上，



不过这取决于你的刷新设置。

在每台服务器上重复安装 MEM 和监控代理，并观察配置面板上的显示结果。图 16-2 显示了配置面板展示每个监控代理报告的实例。你将看到多个图形和数据信息，这些信息包括数据库可用性、连接状态、总的数据库活动状态、查询的响应时间和每个服务器的关键问题列表。

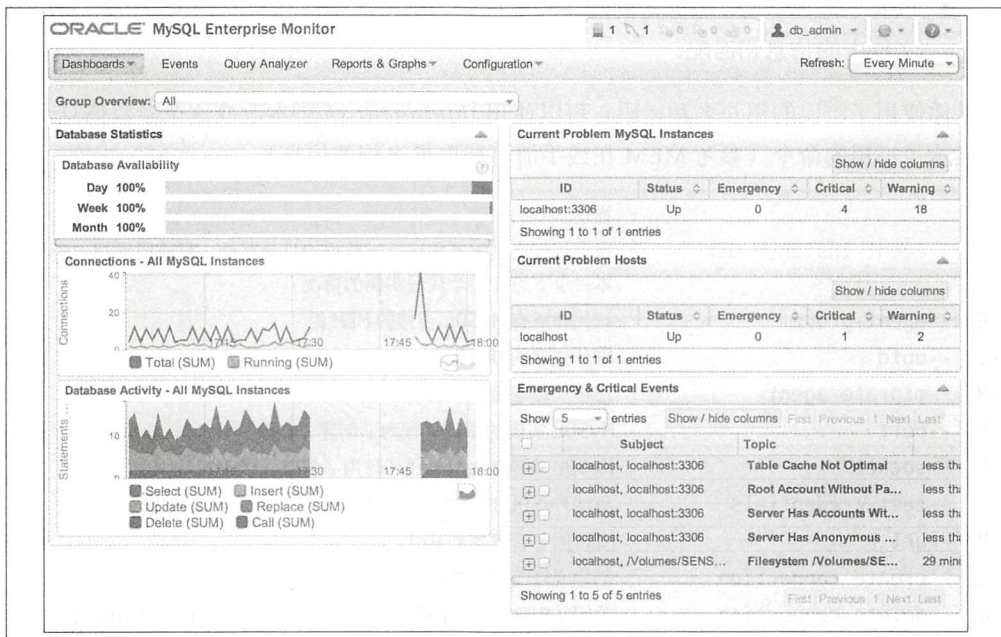


图16-2: MySQL企业监控配置面板

## 解决监控代理的相关问题

虽然安装监控代理的过程很简单,但是有时也会出错,如果提供的信息正确,则安装成功。

下面是用于诊断和更正监控代理连接 MEM 过程中出现的相关问题的流程。

- 如果监控代理开始运行了,但服务器没有被显示在配置面板上,或者如果在热图中代理或服务状态显示有错误(即红色点),那么请检查 `mysql-monitor-agent.log` 文件,其中含有大量有助于解决大部分问题的信息,该日志文件位于以下位置。

Mac OS X

`/Applications/mysql/enterprise/agent/`

Linux/UNIX

`/opt/mysql/enterprise/agent`

Windows

`C:\Program Files\MySQL\Enterprise\Agent`

- 检查监控代理访问 MySQL 服务器的用户账户和权限。
- 核实本地 MySQL 服务器的端口和主机名。确保这些信息与 `\etc\bootstrap.profile` 文件中的信息相匹配。

如果你使用了错误的用户名和密码，可以使用 `\bin\agent.sh` 脚本改变这个参数信息。下面显示了可用的命令（参考 MEM 在线手册可获取更多相关信息）。

选项	描述
-D, --agent-instance-dir	每个代理实例配置和运行时数据的目录路径。只适用于同一个基本目录下运行多个代理实例的情况
-G, --agent-group	该代理所有 MySQL 连接的 MEM 组
-I, --uuid	MEM 代理的 UUID
-M, --migrate-agent	迁移代理连接
-P, --port	MySQL 实例的端口（行为：创建、修改）
-S, --socket	MySQL 实例的 socket（行为：创建、修改）
-T, --test-credentials	测试 MySQL 连接的认证信息
-U, --url	MEM 服务管理器的 url
-c, --create-connection	创建或修改一个 MySQL 连接
-d, --delete-connection	关闭并删除一个 MySQL 连接（必须指定 --connection-id）
-f, --force-plain-stdin	强制使用普通的标准输入（明确输入密码）
-g, --connection-group	创建或修改连接所用的 MEM 组（行为：创建、修改）
-h, --host	MySQL 实例的主机（行为：创建、修改）
--help	输出用法信息
-i, --connection-id	连接 ID（行为：修改、删除）
-j, --admin-user	管理级别用户名（行为：创建、修改）
-k, --general-user	普通级别用户名（行为：创建、修改）
-l, --limited-user	限制级别用户名（行为：创建、修改）
-m, --auto-manage-extra-users	自动创建普通或限制级别用户（行为：创建、修改）
-s, --show	显示这个代理上所有 MySQL 连接的相关信息
-t, --run-collection-tests	发现并试图收集操作系统相关的资源，并转储到标准输出（调试用）
-u, --agent-user	MEM 服务管理器的代理用户名
-v, --version	显示代理及其组件的版本

## 配置面板

当安装并启动至少一个代理后，你可以返回到配置面板，并开始配置你的需求。例如，

可以给你的服务器进行分组。比如，你可能想将一些服务器定义为生产服务器，而将其他服务器定义为开发服务器。

使用设置页中的管理服务器选项可重命名服务器。这样就可以在配置面板上使用更有意义的名称，而不改变真正的服务器主机名。还可以创建组，用来合并相关的服务器。这样在不同的配置面板上显示组时，可以折叠组或者根据需要改变其显示。

## MySQL 企业监控组件

MEM 是服务器连续监控和警告的核心。MySQL 网站上是这样形容它的：“它就像你身边的虚拟 DBA 助手，可以向你推荐减少安全漏洞、提高复制性能、优化性能等的最佳解决方案。”如果不考虑市场营销，MEM 可以提供满足日益增长的业务数据中心需求的专业工具。

MEM 包括以下几个主要功能：

- 监控所有服务器的健康状态，并以单一的网页形式显示。
- 600 多个关于 MySQL 服务器及其运行的操作系统的指标。
- 具有监控性能、复制、模式和安全的的能力。
- 在代理已经安装的情况下，具有监控服务器操作系统统计信息的能力。
- 通过简单的热图显示服务器的实时健康状态。
- 超过指标阈值告警。
- 实施 MySQL 创建者的最佳实践规则集。
- 与 MySQL 的所有最新特性都相符，包括 UUID、GTID 等。
- 自动监测服务器的变化（如复制角色、监测新服务器实例等），并做相应的调整。
- 查找出每个服务器上最关键的问题。
- 趋势、预测和预警功能被添加到图和事件中，以帮助发现潜在的问题。
- 3.0 版本在连接配置面板的用户接口上做了许多改变，这样使得监控服务器变得更容易。

◀ 591

MEM 由运行在内部网络上的分布式 Web 应用程序组成。每个向 Web 服务器组件发送指标的 MySQL 服务器都会安装一个监控代理，即配置面板（Dashboard）。这里有所有描述服务器状态的统计信息和图形。MEM 还包含实施最佳实践的指导顾问，确保服务器的配置合理，并能高性能运行。

## Dashboard

MySQL 企业版监控工具的图形客户端是配置面板（Dashboard），它是运行在监控服务器上的 Web 应用程序。Dashboard 提供了一个监控所有服务器的可用性、安全和性能数





从解释说明中可以看到，至少存在一个没有密码的“root”账号。查找服务器，发现这个问题是真实存在的，而且建议的解决措施是最谨慎的。显然，为了确保安全性，这个问题是必须要解决的。

```
mysql> SELECT user, host, password FROM mysql.user WHERE user = 'root';
```

user	host	password
root	localhost	*[...]
root	127.0.0.1	
root	::1	
root	192.168.1.15	*[...]
root	192.168.1.102	*[...]
root	%	*[...]

6 rows in set (0.00 sec)



我们已经在前面的输出报告中标示出了密码。查询结果实际上显示的是密码列的摘要。

每解决一个问题，接口就刷新一次，将解决的问题从列表中删除。如果有些问题你觉得不需要更改，可以手动从列表中将它们删除。例如，你可能在测试的时候故意设置有问题的发展和测试服务器。也可以添加自己的笔记以供参考。通过这种方式，你可以很快发现网络中那些需要注意的服务器。

# 监控代理

监控代理是一个特殊的轻量级应用，负责收集 MySQL 服务器的相关信息，包括主操作系统的统计信息等。因此，监控代理是监控工具的主要组件，想在每个需要监控的服务器上安装监控代理，就要求这个代理不仅仅是轻量级的，而且几乎是透明的，即监控代理不能带来明显的性能下降。

# advisor

MEM 工具中有一个与典型企业监控方案不同的功能，即监控系统特定区域的性能和配置，并在服务器偏离 MySQL 设计者定义的最佳状态时发出警告。也就是说，可以在系统配置、安全或性能出了问题时立即得到反馈。该功能称为 advisor，可以针对不同方面进行监控和报告，包括：

## Administration

监控通用数据库的管理和性能。

## Agent

检查在你的网络中安装的代理的状态。

## Availability

监控 MySQL 连接和进程的健康状况。

## Backup

检查备份作业、资源和 MySQL 企业备份任务的状态。

## Cluster

MySQL 集群产品中的 advisor。

## 595 Graphing

标识图形元素相关的问题。

## Memory Usage

当次优条件出现时，标识内存使用率的变化并发出告警。

## Monitoring and Support Services

MEM 服务自带的 advisor，它甚至可以建议如何做得更好。

## Operating System

标识 MySQL 服务器上的主机的操作系统中存在的潜在问题。

## Performance

标识实施 MySQL 的性能最佳实践规则时存在的性能差异。

## Query Analysis

标识有关查询分析器的问题。

## Replication

标识特定复制条件下有关配置、健康、同步（延迟）和性能问题。

## Schema

标识数据库和 schema 对象的更改情况。可以监控更改，并在发生不必要或意想不到的变化时发出警报。

## Security

确定潜在的安全漏洞，并发送警报。

每个 advisor 使用最佳实践规则集全面涵盖了服务器某个方面的信息。advisor 有助于识别服务器什么地方需要注意，并给出如何改进或修正当前状态的建议。如果这组 advisor 集还不够全面，可以根据需求创建自己的 advisor。

## 查询分析器

复杂的数据库和应用程序可能需要执行复杂查询。如果使用 SQL 表达式，通常写出来的查询语句效率都不高。而且，写得很差的查询往往会降低性能。专业的数据库管理员都知道这一点，他们通常在进行数据库性能诊断时首先检查查询语句。

一般可以通过查看慢查询的日志或进程列表（如执行 SHOW PROCESSLIST 命令）来发现低性能的查询。一旦发现了需要注意的查询，就可以使用 EXPLAIN 命令查看 MySQL 是如何执行该查询的。虽然这个过程众所周知，而且一度被数据库管理员使用，但这是一个很费力的任务，且不容易写成脚本。随着数据库的复杂性的增加，低性能查询的诊断也越来越费力。

◀ 596

你可以使用这种方法检查用户查询，但是如何审查包含在应用程序代码中 SQL 语句的性能呢？这种情况是最难诊断和修复的，因为它需要更改应用程序。假定可以更改应用程序，如何建议开发者改进查询呢？

不幸的是，很少有人怀疑应用程序代码中的 SQL 语句是影响性能的根源。当遇到性能问题时，DBA 和开发人员都急于责怪服务器或系统，而不是嵌在应用程序中的 SQL 查询。更糟糕的是，MySQL 系统不支持强大的性能指标集，也不支持发现麻烦查询。

如果能找出服务器上所有长时间运行的查询，并检查最慢的查询，可以很好地解决这些问题。企业监控工具的查询分析器可以做到这一点。

可以在 Dashboard 上启动查询分析器。查询分析器的安装和配置需要做一点工作，所以一定要参考入门手册了解详细信息。

查询分析器实时显示查询的性能统计信息。它在一个地方显示来自所有服务器的所有查询的信息，所以不需要逐个服务器地寻找低性能的查询。这个列表还维护了所有查询的历史，这样就不必担心日志的存储空间问题。

每个查询都有两种不同的视图：一个是规范视图（没有数字数据），以图像的方式显示查询；而另一个视图显示查询的具体时间和访问数据。最好的是，advisor 甚至可以提醒你查询什么时候在哪个服务器上执行。

查询分析器可以发现性能不好的查询，然后通过检查可疑查询的执行计划来确定解决方案。显然，光这个功能就可以节省大量时间，特别是在以下情况下：开发或优化部署应用程序，或者需要优化查询以提高性能。

MySQL 企业版还包括专业技术人员的支持，技术支持人员可以帮助你开发、部署和管理你的 MySQL 服务器。技术支持包括问题解决、咨询、访问常见解决方案的在线知识库以及（白金版才有）技术客户经理，他是帮助你解决所有 MySQL 问题的联络人。



这取决于你怎样购买 MySQL 企业版，咨询和额外的协助可能会产生额外的费用。

## 使用 MySQL 企业版监控

前面介绍了 MySQL 企业版的功能及其组件，接下来通过实例看看这些工具是如何使组织机构受益的。在这个例子中，假设基于 Web 的公司已经具备信息基础设施，这并不罕见。这个例子是行业中复制模型的典型代表，包括一个复杂的多 master 复制配置，并支持多个数据库系统之间的复制（只部分复制某些数据库）。

这种情况如图 16-4 所示，由两个通过复制进行连接的数据中心组成，并通过复制实现高可用性和负载均衡。这些数据中心托管公司的数据库。在旧金山的生产数据中心用于日常运作。连接到这个主机的是位于西雅图的用于开发的 slave，这个 slave 主要负责创建和加强产品线。

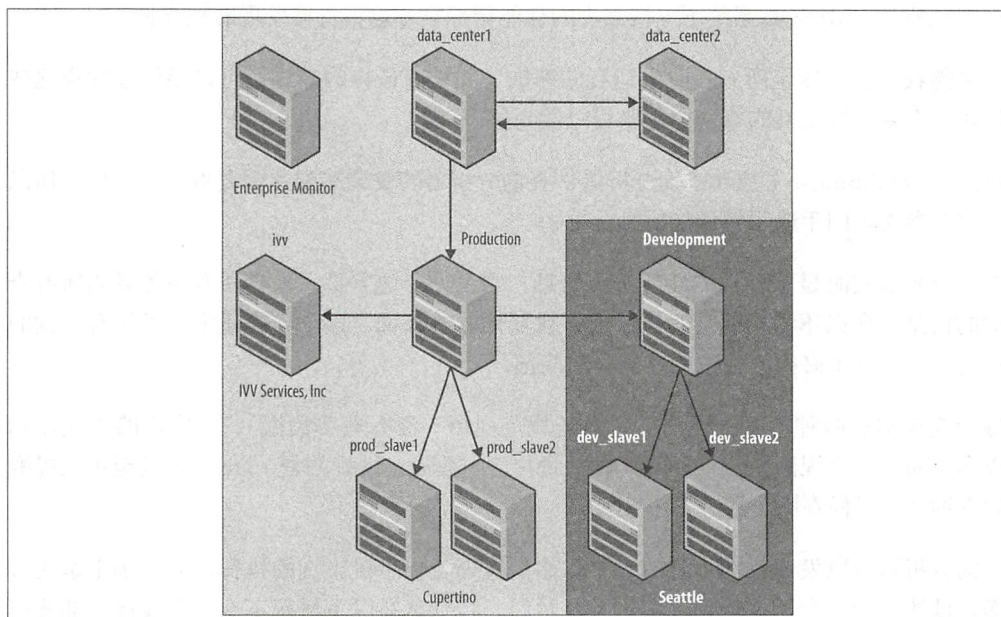


图16-4：信息基础设施示例



数据中心中的每个 slave 都可以（而且一般是这样做的）包含其他未复制的数据库。例如，产品服务器通常承载人力资源数据库，而这些数据不会被复制到大部分 slave 上（例如不会被复制到开发中心）。同样的，第三方服务器承载它自己的结果数据，而开发服务器拥有不同的开发状态下的产品线数据库的各种版本。

这个拓扑结构可以很容易地在 MySQL 企业监控配置面板上可视化。图 16-5 显示了这个示例拓扑是如何在配置面板上查看的。

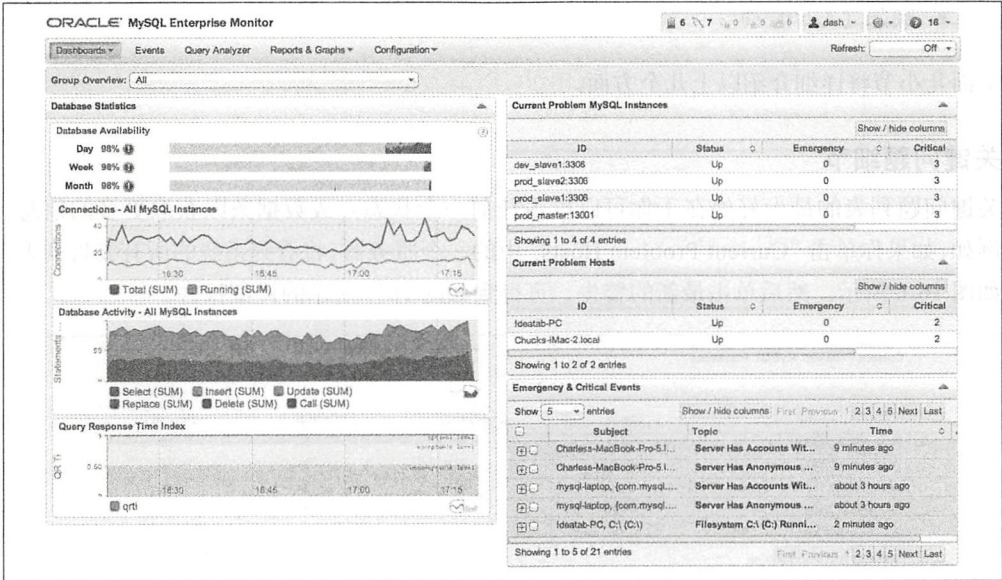


图16-5: 配置面板

图片中显示的服务器拓扑结构是一个基本的复制结构，其中在每个站点上包含了一个 master 和多个 slave。在西雅图的卫星办公室的 master 作为 slave 被连接到旧金山主办公室的 master 上。服务器尚未配置为最佳状态，以通过 MEM 的特征和功能确定最理想的配置和存在的重要问题。

这是配置面板的默认显示方式。正如我们所看到的，这时有报告每个服务器的可用性、分类数据库活动、查询响应时间和潜在问题的长列表。配置面板上还显示了一些其他问题，其中包括参与复制的服务器和所有 MySQL 服务器监控的详情列表。我们将在后面的章节中讲解这些情况的实例。

正如你所想的一样，这个页面提供了服务器上所有的相关信息，因此，可一眼看到你的整个信息架构的健康状况，其中包含监控代理标识出的所有潜在的和重要的问题。显然，没有比这更简单的监控了。

## 监控

MySQL 企业版有几种简化监控的方法来管理复杂的信息基础设施，包括：

- 关键问题细节
- 服务器综合图
- 服务器详情
- 复制详情
- 顾问（Advisor）

600 下面几小节将详细介绍以上几个方面。

### 关键问题细节

关键问题列表的最大好处在于你可以单击任何一个状态点或数据条以获取更多的信息。例如，如果你单击“Current Problem Hosts”中的一个服务器，将看到系统的所有警告列表，如图 16-6 所示。然后单击最新的警告，可得到如图 16-7 所示的详细报告。

Current Problem MySQL Instances

Show / hide columns

ID	Status	Emergency	Critical	Warning
prod_slave2:3306	Up	0	3	13
prod_slave1:3306	Up	0	3	14
prod_master:13001	Up	0	3	11
dev_slave1:3306	Up	0	1	14
dev_master:13001	Up	0	3	7

Showing 1 to 5 of 5 entries

Current Problem Hosts

Show / hide columns

ID	Status	Emergency	Critical	Warning
Ideatab-PC	Up	0	2	0
Chucks-iMac-2.local	Up	0	1	2

Showing 1 to 2 of 2 entries

Emergency & Critical Events

Show 10 entries

Show / hide columns

First Previous 1 2 3 Next Last

<input type="checkbox"/>	Subject	Topic	Time	Actions
<input type="checkbox"/>	mysql-laptop, dev_master:13001	Root Account Without Password	about an hour ago	
<input type="checkbox"/>	mysql-laptop, dev_master:13001	Server Has Accounts Without A ...	about an hour ago	
<input type="checkbox"/>	mysql-laptop, dev_master:13001	Server Has Anonymous Accounts	about an hour ago	
<input type="checkbox"/>	cbell-ubuntu-vm, prod_slave2:3306	Root Account Without Password	8 minutes ago	
<input type="checkbox"/>	cbell-ubuntu-vm, prod_slave2:3306	Server Has Accounts Without A ...	8 minutes ago	
<input type="checkbox"/>	cbell-ubuntu-vm, prod_slave2:3306	Server Has Anonymous Accounts	8 minutes ago	
<input type="checkbox"/>	Charles-iMac.local, prod_slave1:...	Root Account Without Password	9 minutes ago	
<input type="checkbox"/>	Charles-iMac.local, prod_slave1:...	Server Has Accounts Without A ...	9 minutes ago	
<input type="checkbox"/>	Charles-iMac.local, prod_slave1:...	Server Has Anonymous Accounts	9 minutes ago	
<input type="checkbox"/>	Chucks-iMac-2.local, prod_master:...	Root Account Without Password	10 minutes ago	

Showing 1 to 10 of 21 entries

First Previous 1 2 3 Next Last

图16-6：警告列表示例

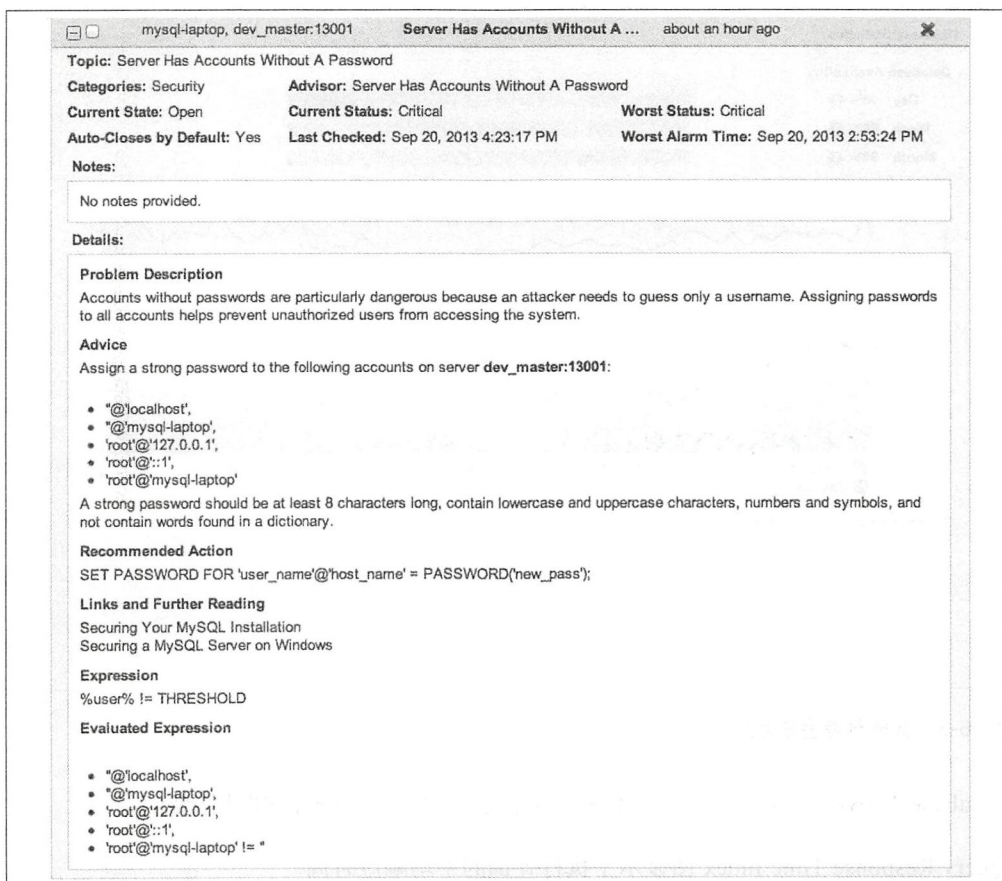


图16-7: 警告报告示例

这份报告显示了发生警告的服务器、警告发生的时间，以及遵循既定的最佳实践的建议。警告报告使 MEM 功能脱颖而出，所以它被称为“虚拟 DBA 助手”。警告使监控更容易，它可以捕获组织机构中的服务器问题，然后在同一个地方显示这些问题。这样节约了时间，也避免了高成本诊断或行为监控的乏味，并提供了如何快速解决问题的提示。

## 服务器综合图

配置面板的左边显示了一个综合图，以代表每个被监控的服务器的数据状态（如图 16-8 所示），其中包括数据库活动、所有服务器的连接状态和查询响应时间。

Database Availability 图显示了有关可用性的统计信息。像所有图表一样，数据是从所有被监控的服务器上收集来的。

Connections 图显示每段时间内所有被监控的服务器的连接时间。



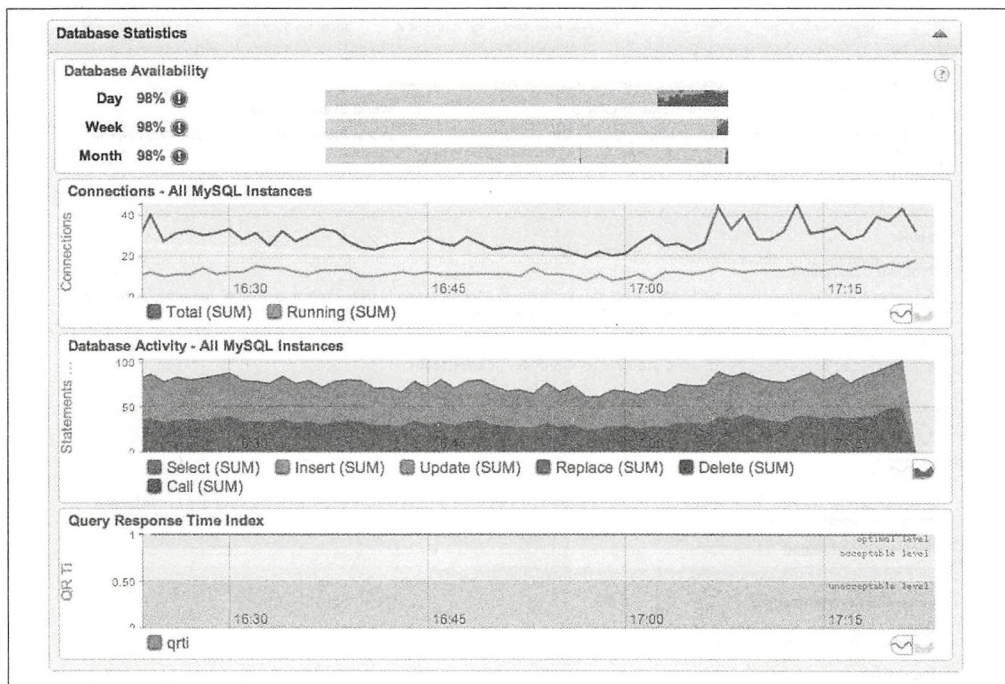


图16-8: 服务器综合图示例

Database Activity 图显示了对查询按照相关性进行分类，并统计其调用次数。

Query Response Time Index 图显示了执行查询的平均响应时间。



你可以通过 Reports and Graphs 选项卡查看更多具体的统计报告。

## 服务器详情

配置面板的另一个很棒的功能是：可以单击服务器列表中某个特定的服务器，查看该服务器系统的更多详情。服务器详情报告中显示了以下信息：MySQL 服务器的版本、MySQL 服务器最后一次启动的时间（运行时间）、数据存放的位置、主机操作系统、CPU、内存大小、磁盘空间大小和网络信息。

可以通过这些信息列一个清单（确定网络上有什么硬盘），还可以快速了解 MySQL 服务器运行在什么操作系统上，从而为解决问题提供线索。例如，要远程登录某个服务器解决问题，登录服务器之前需要知道它的主机名、IP 地址、MySQL 版本，以及主操作系



统信息，管理员需要记下所有重要的信息。图 16-9 所示的是配置面板上的服务器详情的样例。

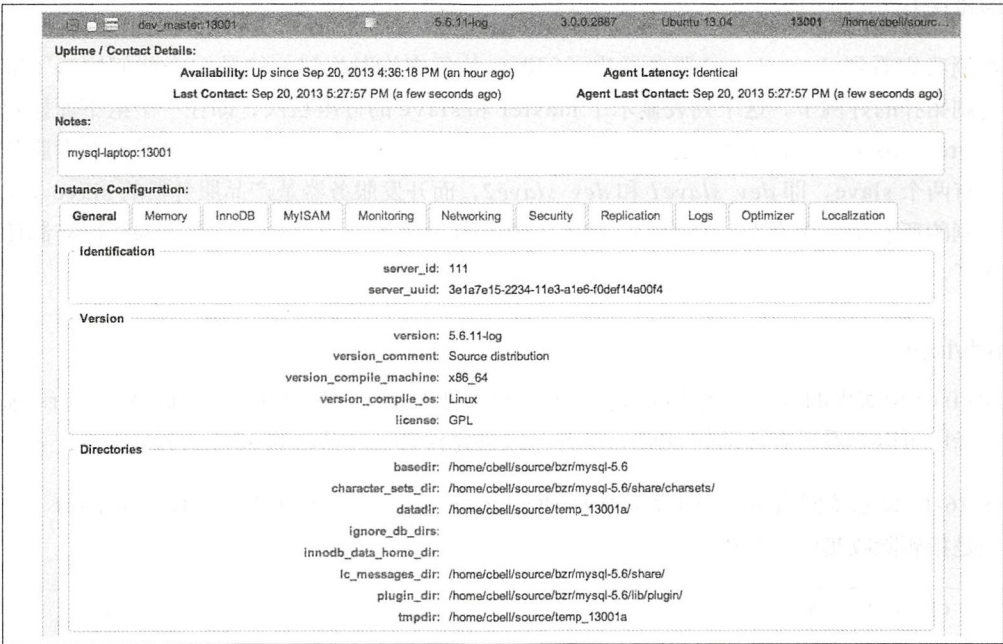


图16-9：服务器详情

## 复制详情

配置面板上的“复制”选项卡包括参与复制的所有服务器的列表。这些信息以列表的形式呈现，与 MEM 中的所有列表一样，你可以单击每个选项以获取更多的信息。图 16-10 显示了复制详情报告的一个示例。

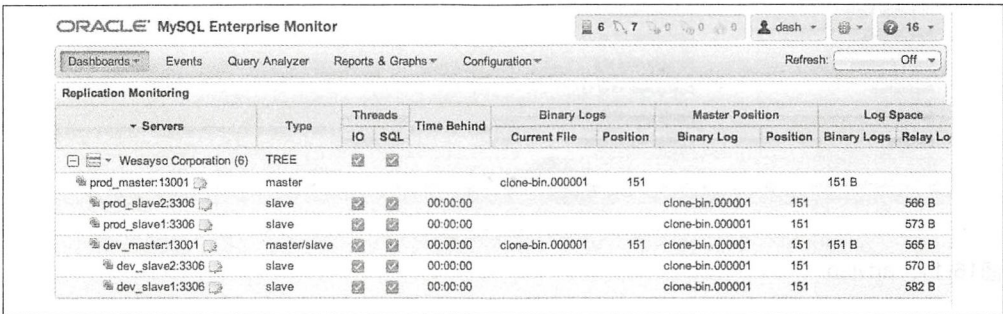


图16-10：复制详情

请注意，列表中的项目是按照拓扑结构分组的（如“Wesayso Corporation”，可以重命名），

这些项目包括拓扑类型、服务器执行的角色，以及复制相关的重要统计信息（包括线程状态、比 master 滞后的时间、当前二进制日志、日志位置、master 日志信息和最近发生的错误）。

本例我们看到 `dev_slave2` 服务器出了问题，执行查询时出错。这是一个如何快速了解复制拓扑的好例子。这个列表显示了 master 和 slave 的分组层次，即在一个组下面显示 master，隶属于 master 的 slave 显示在 master 下面。从图 16-10 中很容易发现开发服务器有两个 slave，即 `dev_slave1` 和 `dev_slave2`，而开发服务器是产品服务器的 slave。将复制的所有信息保存在一个地方，就不需要在每个服务器上单独执行烦琐的监控复制任务了。

### advisor

advisor 中实现的最佳实践使得所有的警告和图更加有益。从配置面板上的 Advisors 选项卡中可以查看所有活动的 advisor（并可以创建你自己的活动的 advisor）。

图 16-11 显示了网络中某个服务器的活动 advisor，可以启动、禁用或取消任何 advisor（取消是指删除收集到的数据）。

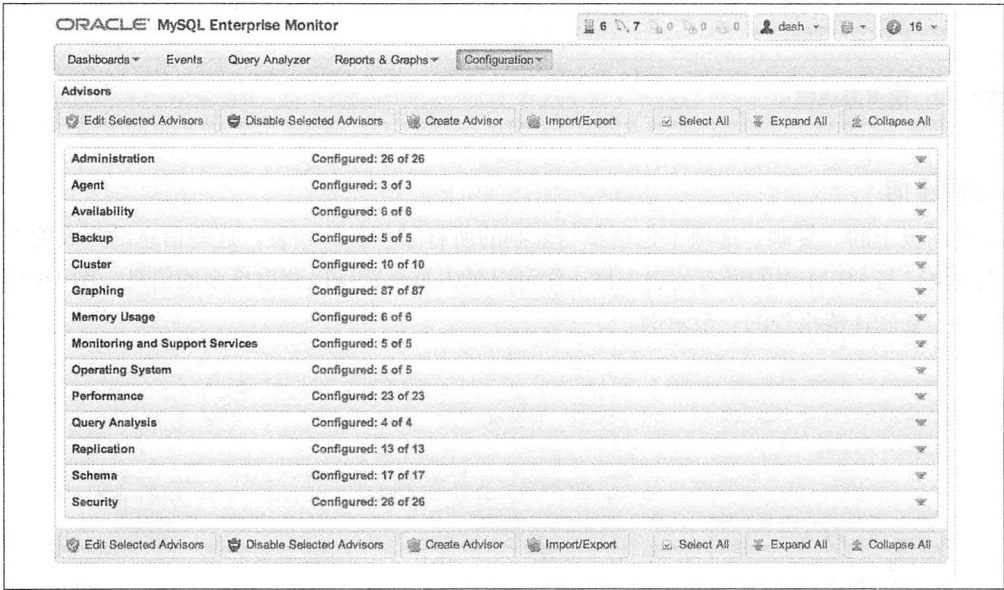


图16-11: advisor

也许该页中最有用的功能是添加自己的 advisor。这个功能允许你根据特定的需求扩展 MEM。另外，从手动监控方案转到监控工具上时，它还可以为你提供一个最有用的功能，即保留你的辛勤劳动成果。

例如，如果你创建了一个用于监控自定义应用的报告机制，可以为它创建一个 advisor，并将警告添加到配置面板。具体如何添加新的 advisor 和警告，在 MySQL 企业网站中的 MySQL 企业监控手册中有详细描述。这个自定义功能是 MySQL 企业版中最强大但却未被充分利用的功能之一。

## 查询分析器

MEM 的最强大功能之一是查询分析器。查询分析器的工作原理是：使用 MySQL 代理拦截并处理 SQL 命令，然后将这些 SQL 命令传递给本地服务器执行。它可以记录统计数据，以便随时查看这些信息。查询分析器还支持 advisor，这个 advisor 对慢查询发出警告。图 16-12 显示了 MySQL 代理拦截查询并向 MEM 报告统计信息的概念图。



新版本的查询分析器可以使用性能架构的数据库来收集统计信息，而不用使用 MySQL 代理。

606

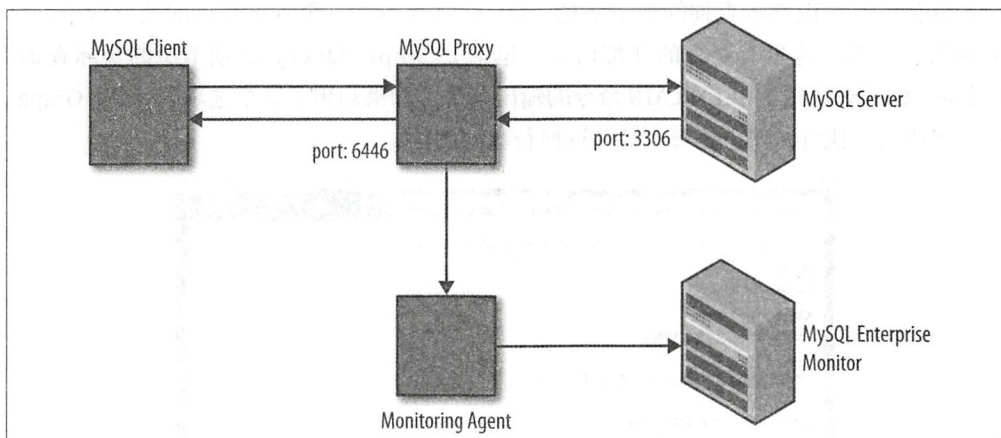


图16-12：使用MySQL代理为查询分析器收集数据



查询分析器在自定义的 6446 端口（默认）运行，而且可能会带来性能延迟。因此，你应该仅在查找问题时启用它。

查询分析器和配置面板的一个好处在于工具之间的紧集成。如果你有一个慢查询的警告，或者需要深入研究 CPU 利用率或其他 MySQL 统计信息的报告，将看到查询分析器页面（单击“查询分析器”选项卡可以直接进入这个页面）。图 16-13 所示的是配置面板的查询分析器页面。



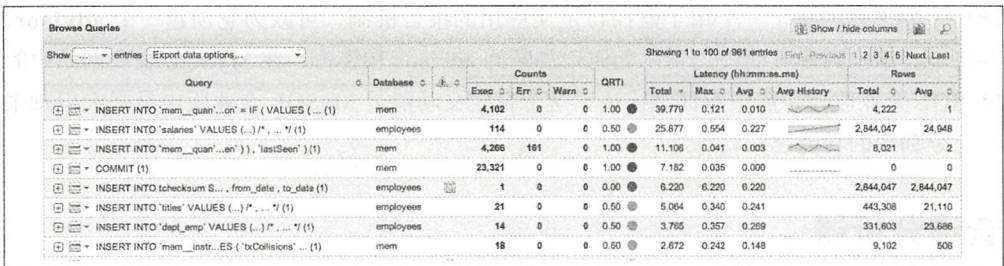


图16-13: 查询分析器页面

607 查询分析器页面的左边显示了服务器列表。单击某个服务器，将看到该服务器上执行的查询列表，按查询运行的时间降序排列。还可以使用顶部的图缩小时间范围，以查看特定时间段内的查询。

单击列标题可以排序，便于发现重复的查询。可以按查询运行的时间、查询语句和操作的的数据量等对行进行排序。

单击任何一行可以获取查询的细节信息。图 16-14 显示了一个查询报告的典型例子。与其他报告一样，这里有更多的详细信息，包括 Example Query 选项卡中的实际查询，Explain Query 选项卡中 EXPLAIN 命令的输出结果（如果启用了这个选项），以及 Graphs 选项卡中关于执行时间、执行次数和返回行数的图等。

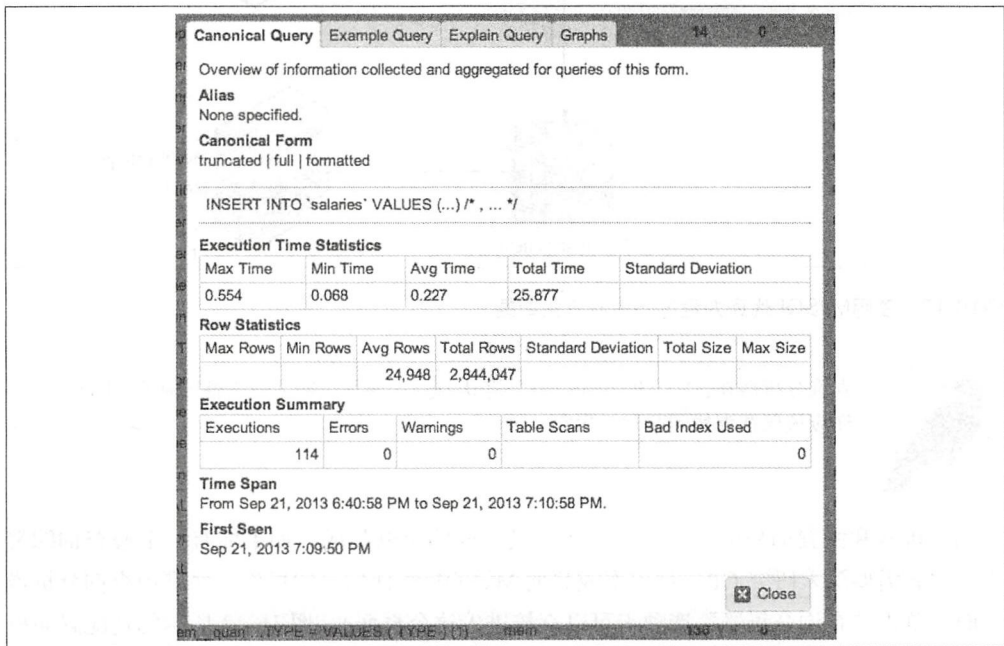


图16-14: 查询报告



## MEM 和云计算

MySQL 企业工具在云计算环境中也很有用。从数据和服务器实例的持久性来说规则相同，只是要为 MEM 服务器设置一个固定的 IP 地址。重复启动和停止 MySQL 实例将不会造成不良影响，但是更改主机名及某些重新配置可能导致监控代理停止报告任何信息。每个服务器的唯一 ID 存储在 *mysql.inventory* 表中。虽然改变了 IP 地址，它也不会被重复添加到这个表中，因为这个服务器的化身名称（相同的 ID 名称）已被存储在这个表中。因此，MEM 可能无法与该服务器联系，因为它可能使用陈旧的数据（旧 IP 地址）。一般截断 *mysql.inventory* 表可以解决这个问题（因为这个服务器将被再次添加到这个表中），但是最好为所有的服务器使用相同的主机名和 IP 地址。

在商业提供者的云上运行 MySQL 企业版有一个明显的好处：只需要为计算和存储付费。云内的数据传输通常是免费的，或者其费用比将数据从云中导入和导出低得多。

这个报告显示了捕获到的查询详情，包括查询的规范形式（即书面查询的图形表示）及其执行详情，例如执行时间和返回或受影响的行。

608

我们再一次看到了这样的监控工具：它可以为 MySQL 问题诊断节省大量的时间。MySQL 企业版的查询分析器组件是一个重要的监控工具，有助于维持服务器的正常运行和查询的高效执行。

## 更多信息

本书不对 MEM 做全面而详细的描述，这些信息可以从网上获取。要想了解更多关于自定义 advisor 的信息，请参看在线 MySQL 企业监控手册。如果你是 Oracle 客户，可以查看 Oracle 技术支持网站上的常见问题解答。

## 小结

609

MySQL 服务器已经成为最流行的开源数据库系统。它的用途相当广泛，任何组织机构都可以使用它，从一个为用户提供信息的独立的 Web 服务器，到在线联机事务处理系统（OLTPS），再到高可用的不断扩展的大规模数据中心。MySQL 可以处理所有这一切。而 MEM 将 MySQL 扩展到大型应用上，并保存了 MySQL 的最佳性能和可靠性。

如果你需要高可用性并希望建立最好的和最可靠的基于 MySQL 的数据中心，那么就应该考虑购买 MySQL 企业白金版。或许你还有其他选择，但是没有哪个比 MySQL 企业

版更专业，也没有哪个能够以便宜的价格为你提供成熟的 advisor 和查询分析器工具。

Joel 准备好了。他已经把提案发出去了。他建议购买 MySQL 企业版来管理公司所有的服务器。Summerson 先生以削减开支闻名，所以 Joel 已经准备好捍卫自己的建议，决定不放弃，直到 Summerson 先生听进他的建议为止。

每当听到脚步声临近时，Joel 都会看看门口，他知道老板随时会来。一阵脚步声响起，Joel 再一次紧张起来。但 Summerson 先生连看都没看一眼就快速走开了。

“请等一下。” Joel 低声说。

“Joel，我喜欢 MEM 这玩意儿。我希望你评估一下整体成本和……”

Joel 打断老板的话：“我已经给你发了一份报告，Bob。”

Summerson 先生扬起眉毛说：“把它发给我，我看看是否可以把它列入明年的预算中。”然后 Summerson 先生就离开了，走向下一个受害者。

Joel 回到座位上，得意地把双臂叉在胸前。“我看那是赞许吧。”他说。

# 使用MySQL实用工具管理 MySQL复制

Joel 对这份工作越来越得心应手。过去几个月他完成了很多任务，包括设置复制和 MySQL 调优。最近，他的老板同意让 Joel 再招一个 MySQL 管理员。没错，Joel 终于可以自己招人。

一阵熟悉的敲门声之后，老板又一次突然闯入，扰乱了 Joel 的狂欢。

“Joel，我有几个候选人想面试一下。在那之前，我需要你草拟一个训练手册，确保将所有的窍门和方法，还有你那些管理技巧都写进去，这样我们的新员工就能很快跟上进度。就从复制开始吧，它让我很操心。”

没等 Joel 回复，老板就消失了，又去安排下一个任务了。Joel 笑了，把这当成一次挑战。

突然，老板把头探回门口说，“一定要用 DocBook 格式啊，搞文档那帮人就喜欢这个格式。”

“太好了！”老板一离开，Joel 就嘲笑着说。“看来我得去趟文档小组，看看他们都用什么工具编译 DocBook 文件。”

MySQL 复制是一个非常完善和可靠的服务器功能。从前面章节中我们已经知道，启动和配置复制需要一定的知识，以及一些根据具体需求而定的步骤。管理 MySQL 复制的关键是知道什么时候该做什么。

很多管理 MySQL 复制的最佳实践都已经在前面的章节中介绍过，大部分是在高可用解决方案中讲述学习和应用复制属性。然而，我们觉得要专门写一章来总结与复制有关的

常见任务，这样才更加完整。此外，我们还特别讲述了两个重要的工具，不需要第三方工具或复杂的安装步骤，就能够立即获得 MySQL 的高可用性。

这一章，我们将讲述复制管理的核心任务、最佳实践，以及管理多个复制服务器的工具，包括执行故障转移和切换的工具。首先我们简单介绍一下复制管理员面临的常见任务。

## 常见的 MySQL 复制任务

幸好管理一个复制装置非常容易。有几件事情必须定期做，但很少需要更改或调整复制配置。这一小节总结了管理 MySQL 复制的时候可能遇到的常见任务。

我们忽略了一个显而易见的任务，即搭建复制，可以参考第 3 章中搭建复制的完整指南。不过，我们会介绍一个搭建复置的工具，只需一个命令即可实现。

### 状态检查

首先我们来看最可能执行的最常见的复制管理任务：检查 slave 的状态，确定是否有错误或者其他异常。这是检查复制状态的基本方法。主要命令是 `SHOW SLAVE STATUS`。这个命令列出一大串关于 slave 的信息。输出见示例 17-1。

示例17-1: `SHOW SLAVE STATUS` 示例

```
mysql> SHOW SLAVE STATUS \G
```

```
***** 1. row *****
```

```
Slave_IO_State: Waiting for master to send event ❶
Master_Host: localhost ❷
Master_User: rpl ❸
Master_Port: 13001 ❹
Connect_Retry: 60
Master_Log_File: clone-bin.000001 ❺
Read_Master_Log_Pos: 594 ❻
Relay_Log_File: clone-relay-bin.000002
Relay_Log_Pos: 804
Relay_Master_Log_File: clone-bin.000001
Slave_IO_Running: Yes ❼
Slave_SQL_Running: Yes ❽
Replicate_Do_DB:
Replicate_Ignore_DB:
Replicate_Do_Table:
Replicate_Ignore_Table:
Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
Last_Errno: 0 ❾
```

613



```

        Last_Error: 10
        Skip_Counter: 0
    Exec_Master_Log_Pos: 594 11
        Relay_Log_Space: 1008
        Until_Condition: None
        Until_Log_File:
        Until_Log_Pos: 0
    Master_SSL_Allowed: No
    Master_SSL_CA_File:
    Master_SSL_CA_Path:
    Master_SSL_Cert:
    Master_SSL_Cipher:
    Master_SSL_Key:
    Seconds_Behind_Master: 0 12
Master_SSL_Verify_Server_Cert: No
        Last_IO_Errno: 0 13
        Last_IO_Error: 14
        Last_SQL_Errno: 0 15
        Last_SQL_Error: 16
Replicate_Ignore_Server_Ids:
    Master_Server_Id: 101 17
        Master_UUID: aa73ae8c-2a9d-11e2-936b-7e5b4e755f7e 18
    Master_Info_File: /Volumes/Source/source/temp_13002/master.info
        SQL_Delay: 0 19
    SQL_Remaining_Delay: NULL 20
Slave_SQL_Running_State: Slave has read all relay log; waiting
                        for the slave 21 22
                        I/O thread to update it
    Master_Retry_Count: 86400
    Master_Bind:
    Last_IO_Error_Timestamp: 23
    Last_SQL_Error_Timestamp: 24
    Master_SSL_Crl:
    Master_SSL_Crlpath:
    Retrieved_Gtid_Set: AA73AE8C-2A9D-11E2-936B-7E5B4E755F7E:1-2 25
    Executed_Gtid_Set: A73AE8C-2A9D-11E2-936B-7E5B4E755F7E:1-2, 26
                        BB8BF918-2A9D-11E2-936B-C5790DD70EC1:1

```

这个例子包含很多行，在检查 slave 状态的时候需要检查其中几个重要的行。下面我们将这些地方分组讨论：

② ③ 这几行检查 master 的连接信息，以确保 slave 连接的是正确的 master。

④ ⑤

⑥ ⑦

- 614 ➤ 9 10 这几行检查错误。所有这些行都应该是 0(表示错误个数),或为空(表示错误信息),  
13 14 或不包含任何关于错误的描述。  
15 16  
22 23  
24
- 1 7 这几行检查状态信息。从前面几章我们知道,复制中存在很多不同的状态。通常,  
8 21 要确保没有哪个状态表明 slave 出错或者丢失了与 master 的连接。
- 11 12 这几行检查 slave 延迟,确定 slave 离 master 有多远(秒,事务等)。这取决于具体  
18 19 的复制装置,有些延迟是可接受的。  
20
- 18 25 这几行检查 GTID 的问题。这些信息用于诊断 slave 上丢失的事务。  
26

更多详情请看在线参考手册,包括每一行是什么意思,以及一个健康的 slave 应该是什么值。

除了 slave 状态以外,还可以从 SHOW PROCESSLIST 的输出中收集 master 的信息。示例 17-2 给出了在 master 上运行这个命令的片段,这是一个健康 slave 连接的典型状态,由 master 上的 dump 线程表示。

示例17-2: SHOW PROCESSLIST片段

```
mysql> SHOW PROCESSLIST \G
```

```
***** 1. row *****
      Id: 4
      User: rpl
      Host: localhost:51234
      db: NULL
      Command: Binlog Dump GTID
      Time: 2623
      State: Master has sent all binlog to slave; waiting for binlog to be updated
      Info: NULL
***** 2. row *****
      Id: 7
      User: rpl
      Host: localhost:51374
      db: NULL
      Command: Binlog Dump GTID
      Time: 4
      State: Master has sent all binlog to slave; waiting for binlog to be updated
      Info: NULL
[...]
```

615 ➤ 在这个例子中,可以看到对于每个已连接的 slave,都有一行 binlog dump 线程记录。这里,

master 上连接有两个 slave，它们都很健康，因为 state 字段没有提示任何错误。

这些命令能够很好地提供 slave 及其连接的全部细节，而且易于通过 mysql 客户端运行。但是，如果有几十个甚至几百个 slave，或者是带有中间 master 和上千个 slave 的层次型拓扑，那会怎样呢？在每个 slave 上都运行这些命令是非常冗繁的。

幸好，可以使用一些工具让事情变得更加简单。对于那些拥有很多 slave 和复杂拓扑的配置来说，MySQL 企业监控器（MySQL Enterprise Monitor）解决方案是提供一站式监控功能的最佳选择。

当然，也有其他工具。即使有很多 slave，可能你只需要检查其中几个，或者检查某个特定 master 的 slave。这时，可以使用 *mysqlrpladmin* 实用工具的 health 命令，为 master 及其 slave 生成一份健康报告。本章稍后我们会介绍这个实用工具。

## 停止复制

下一个最常见的复制管理任务是停止和重启复制。我们建议任何需要对 slave 做什么的时候都要做这个任务，比如出于维护原因，或更正数据中的错误，或应用程序特定的更改操作。

启动和停止复制的命令包括重置 master 或 slave 的复制配置。我们已经在前面的章节中见过这些命令，为了方便，在这里总结一下（为了清晰起见，将变种作为独立的条目列举出来）：

### STOP SLAVE

停止 slave 上的复制线程(SQL、IO)。当想要断开 slave 的连接、不再接收更新的时候，使用这个命令。

### STOP SLAVE TO\_THREAD

仅停止复制 IO 线程。当希望停止接收来自 master 的更新、但却仍然希望 slave 中继日志继续处理事件的时候，使用这个命令。

### STOP SLAVE SQL\_THREAD

仅停止复制 SQL 线程。当希望中继日志停止处理事件、但却不想停止接收来自 master 的更新时，使用这个命令。

### START SLAVE

启动 slave 的复制线程。

#### 616 > START SLAVE UNTIL SQL\_BEFORE\_GTIDS *gtid\_set*

启动复制线程，直到在碰到 *gtid\_set* 中的 GTID 之前两个线程都到达第一个 GTID，这时线程被停止。

#### START SLAVE UNTIL SQL\_AFTER\_GTIDS *gtid\_Set*

启动复制线程，直到在碰到 *gtid\_set* 中的 GTID 之前两个线程都到达最后一个 GTID，这时线程被停止。

#### START SLAVE UNTIL MASTER\_LOG\_FILE='file', MASTER\_LOG\_POS=*position*

启动复制 slave，直到 master 的日志文件和位置都已达到，之后 slave 线程被停止。

#### START SLAVE UNTIL RELAY\_LOG\_FILE='file', RELAY\_LOG\_POS=*position*

启动复制 slave，直到 slave 的中继日志文件和位置都已达到，之后 slave 线程被停止。

#### RESET SLAVE

该命令导致 slave 删除连接到 master 的设置信息，用于干净地启动复制（例如将 slave 从一个 master 转移到另一个 master 的时候，又称为剪枝，pruning）。这个命令还会启动一个新的中继日志，中断任何对已有中继日志的读操作。

#### RESET MASTER

该命令由 master 发出。用于删除所有的二进制日志文件和二进制日志索引，并创建一个新的二进制日志和索引。显然，这个命令很少使用，而且在常规的复制管理过程中从不使用。例如，如果一个 master 上连接有活动的 slave，那么在这个 master 上运行该命令会中断 slave（可能还会导致数据丢失或不一致）。你可能会用到这个命令的场合是，当 master 已经被离线（或失效），之后你又想让它作为另一个 master 的 slave 重新工作。



当以 SQL\_THREAD\_GTIDS 或 SQL\_AFTER\_GTIDS 使用 START SLAVE UNTIL 命令的时候，可以指定线程的类型。还可以使用 SQL\_THREAD 选项使 START SLAVE UNTIL 格式的命令仅对 SQL 线程有效。若想得到 START SLAVE 命令的完整选项列表，请参看在线参考手册。

如果需要停止、启动或重置大量的 slave，如果不写脚本重复执行一组命令的话，连接每个 slave 是一件很费时间的事情。幸运的是，有一个工具可以实现现在一组 slave 上发出这些命令。本章稍后将介绍 *mysqlrpladmin* 实用工具。

#### 617 > 添加 slave

除了检查 slave 的健康状态以及在 slave 上例行启动和停止复制以外，下一个最常见的复



制管理任务是向拓扑中添加 slave。这么做可能是为了横向扩展 (scale out)，或者使数据变成远程可访问，或者是常规维护过程，即删除 slave，系统变更，然后再将 slave 添加回拓扑。

这个操作最重要的元素，同时也是难倒大多数新手管理员的是，让 slave 与 master 同步。当 slave 的数据库状态较旧的时候，情况尤其复杂。虽然本质上过程是一样的，但是如果新加的 slave 不包含数据的话，会更容易一些（只考虑数据加载选项）。

将 slave 加入拓扑之前，必须考虑 master 上是否存在这个 slave 的较旧状态。如果有，你有以下几种选择。

- 使用 master 上的副本重写数据。通常是将 master 上的备份文件应用到 slave。
- 尝试从最近已知的 master 二进制日志和位置前滚 master 的二进制日志。这种技术称为即时恢复 (point-in-time recovery, PITR)，已经在其他章节提到过。根据 slave 落后的程度不同，这种方法可能比等待 slave 赶上来要快。注意，这种方法需要知道 slave 离开拓扑时，master 的二进制日志文件和位置。
- 连接到 master，并等待 slave 赶上来。



对于那些使用 `--master-info-repo=FILE` 选项启动的 slave 来说，从 master 信息文件 `master.info` 中可以知道 slave 的最近已知 master 二进制文件和位置。对于那些使用 `--master-info-repo=TABLE` 选项启动的 slave 来说，可以查询 `mysql.slave_master_info` 表。

添加 slave 的过程包括决定如何设置 slave 上的数据，然后使用备份、文件副本或者其他形式的数据恢复来执行计划。一旦 slave 上的数据与 master 上的数据接近或一致，就能够以常规的方式建立复制了，如第 3 章所述。为了叙述的完整性，下面总结了具体步骤：

1. 如果没有使用 GTID，确保新 slave 的 `--server-id` 被设置成其中一个 master 或 slave 的唯一标识符。
2. 在 slave 上准备数据。
3. 在 slave 上执行 `CHANGE MASTER` 命令。如果用了 GTID，使用 `MASTER_AUTO_POSITION=1` 代替 master 二进制日志文件和位置。
4. 在 slave 上执行 `START SLAVE` 命令。
5. 使用 `SHOW SLAVE STATUS` 检查 slave 是否有错误。

以上过程称为 slave 自动配置 (slave provisioning)。虽然无法通过单个命令实现所有这些步骤，但是有些实用工具能够使这些步骤更加简单。稍后我们将介绍这些实用工具的实战。

# MySQL 实用工具

Oracle 拥有一套非常有用的管理工具，称为 MySQL 实用工具（MySQL Utilities），这是 MySQL 工作台的一个子产品。实用工具是用纯 Python 编写的，全部都是基于命令行的。

MySQL 实用工具中有很多管理复制的有用工具。前面讨论的很多任务都可以通过这些工具完成。除了复制相关的工具以外，还有很多工具能够用于涉及复制的各种场合。

下面将介绍可供使用的实用工具。我们并不会仔细讨论每个工具的每个功能及每个选项，而是介绍这些工具的用途，并讨论如何最佳使用它们完成复制。如果需要了解这些工具的具体信息，比如可用的选项、功能及限制，请参考 MySQL 实用工具的在线文档。该文档中包含了大量的示例以及对于所有选项的深入描述。

大部分工具都有一个详细信息选项，即 `--verbose` 或 `-v`，用于输出详细信息，可以多次指定。通常，最详细的形式为 `--verbose --verbose --verbose` 或 `-vvv`，表示三倍冗长信息。

## 入门

MySQL 实用工具可以直接从 MySQL 下载站点下载获得，访问 MySQL 开发者板块的下载页面即可。你会看到适合几种常用平台的版本下载，包括 `.tar` 和 `.zip` 文件，它们使用传统的 Python 命令 `python ./setup.py install` 命令安装使用工具。

安装和使用这些工具的唯一要求是使用 Python 2.7 及更高的版本以及 Connector/Python 数据库连接器。Connector/Python 也可在下载页面下载。连接器必须在运行工具之前安装，不过也可以在安装 MySQL 使用工具之后再安装连接器。

## 不通过工作台使用实用工具

要在工作台之外使用 MySQL 实用工具，打开任意终端（控制台）即可，如果还没指定环境变量或 profile，首先要设置 `PYTHONPATH` 环境变量。该变量包含 MySQL 实用工具和 Connector/Python 所在的位置。如果安装的是平台特定的版本，或者通过标准 Python 安装命令安装的，那么产品安装在文件夹 `PythonXX` 下。

## 通过工作台使用实用工具

如果下载和安装的工作台是版本 6 及以后的，并且也安装了 MySQL 实用工具，就可以从工作台打开 MySQL 实用工具。MySQL 工作台 6 以前的版本中包含了 MySQL 实用工具，而 6 及以后的版本不包含。

要从工作台打开 MySQL 实用工具，首先打开工作台，然后单击 Plugins 菜单，选择“启动 MySQL 实用工具的 Shell”项，或者单击多工具图标。可能需要拓宽显示视图，或者单击向右箭头图标，就能看到实用工具的插件图标了。图 17-1 将这两种打开 MySQL 实用工具的方法高亮表示出来了。



图 17-1：从工作台打开MySQL实用工具

启动插件的时候，会打开一个终端（控制台），并将可用的实用工具列表输出出来。图 17-2 展示了启动实用工具插件的结果。



通常，MySQL 实用工具和工作台的开发会导致一年中多次出现新版本。请确保下载页面中 MySQL 实用工具和工作台的版本最新。



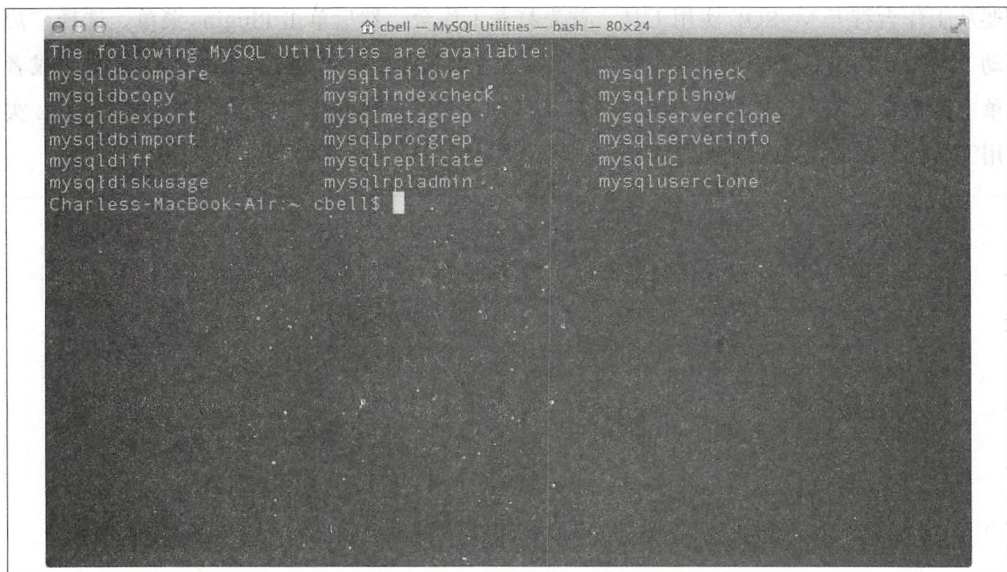


图 17-2: MySQL实用工具插件的控制台

## 621 常用工具

这一小节简单介绍一下那些通用的实用工具。我们将描述每个工具，给出运行每个工具的简单示例，并提供关于这些工具如何用来管理复制的想法。



有些实用工具对某些选项提供了快捷方式，比如 `-s`，而所有工具都能使用前缀匹配选项名，只要这个前缀唯一匹配单个选项即可。例如，假设某个工具有一个名为 `--discover-slaves-login` 的选项，并且没有其他选择与它的前缀相似。那么就可以使用 `--discover=` 或 `--disco` 而无须写完整的选项名。显然，这在输入很多选项的时候非常节省时间。

## 比较数据库的一致性：mysqlbcompare

尽管在一个管理完善的基础架构中，数据不一致很少见，但数据还是有可能损坏的。但是，可能不只是数据出现不一致，对象定义本身也可能不一致。幸运的是，比较数据库的实用工具能够同时比对定义和数据。

在复制中，不大可能遇到对象定义不一致的情况。很多时候，这是用户错误造成的，但也可能是在复制过程中遇到的错误导致的。我们需要一种实用工具能够比较两个表，以确定数据中究竟有何不同。



意外或错误变更可能导致 slave 上的数据不再与 master 同步。例如,如果用户(或应用程序)向 slave 写数据,实际上就带来了不一致。根据变更的类型不同,有段时间可能发现不了这种不一致。

如果在变更从 master 转移到 slave 的过程中出现错误,或者事件执行的时候出现无关故障,也会出现数据不一致。大部分时候,这种错误会被立即发现,但却很难确定哪里不一致甚至是否存在不一致。与其指望数据在 slave 恢复以后保持一致性,进行不一致性检查总是个好办法。

检查不一致性并不是一件容易的事情。此外,有很多确定不一致性的方法。*mysqldbcompare* 工具按行构建校验和,然后将它们分块,对比这些块,确定哪个块包含不一致。虽然这种方法能够有效地发现具体哪些行不一致,但是由于生成校验和的开销很大,对多个数据库或含有大量行的多个表运行这个工具可能会增加运行时间。



由于这个实用工具的运行开销很大,比如它会在表上长时间加锁,所以我们将它视为离线工具。如果想在运行工具的同时保持数据库对用户可用,那么试着在响应时间要求不高的情况下运行,这样就可以用这个资源密集型工具拖延数据库。

622

## 用于复制

*mysqldbcompare* 用于在 slave 上从数据损坏问题中进行恢复。因为这个工具要求对要比较的表加锁,所以它是一个离线工具。

为了将对复制应用程序和用户的影响减小到最小,最好是从活动可行的 slave 中选择一个 slave,然后比较已知的健康 slave 和恢复后的 slave 的数据库。不需要将服务器离线,但是最好在执行这个对比之前将复制挂起(suspend),以避免对比的过程中发生的变更被锁阻碍。

这个工具对于恢复有两个用处。第一,可以使用这个工具比较某个 slave 和另一个 slave 的数据问题,确定哪些地方发生了变更。第二,使用这个工具确定数据恢复的尝试是否成功。

一旦识别到哪些行不一致,就可以开始更正这些不一致。具体方法有很多,从在 slave 上手动变更行,到运行诸如备份或即时恢复这样的数据恢复过程。

这个工具对于恢复数据的不一致尤为有用的一个功能是:它能够生成使数据库同步的 SQL 命令。这个命令能用于在 slave 上更正数据的不一致。

mysqldbcompare 示例

下面的例子在两台服务器上对比数据库，即复制拓扑中的两个 slave。第一个例子在 server2 上发现丢失的视图。这对 slave 来说可能是没问题的。有些管理员会为某些应用程序功能创建视图。注意，这样就会导致数据不一致。因为我们在两个 slave 上运行实用工具，所以期望这两个 slave 完全一致，但显然 supplier 表 id2 所在的行在两个 slave 上不一样。好消息是，现在我们知道该看哪一行，以确定需要做些什么以使两个 slave 上的数据保持一致。

```
$ mysqldbcompare --server1=root@slave1 --server2=root@slave2 \  
inventory:inventory --run-all-tests  
# slave1 上的服务器：……已连接  
# slave2 上的服务器：……已连接  
# 检查 server1 和 server2 上的 inventory 数据库  
WARNING: Objects in server1:inventory but not in server2:inventory:  
VIEW: finishing_up  
VIEW: cleaning
```

623

Type	Object Name	Defn Diff	Row Count	Data Check
TABLE	supplier	pass	FAIL	FAIL
Data differences found among rows:				
--- inventory.supplier				
+++ inventory.supplier				
@@ -1,2 +1,2 @@				
code,name				
-2,Never Enough Inc.				
+2,Wesayso Corporation				

下一个例子，仍然运行同样的命令，不过这次我们会生成同步数据库所需的数据转换语句，并更正数据的不一致。跟上一个例子比较一下。注意，语句列在代码的最后。下面是在 slave 上从数据不一致中恢复的语句。

```
$ mysqldbcompare --server1=root@slave1 --server2=root@slave2 \  
inventory:inventory --run-all-tests --difftype=sql  
# slave1 上的服务器：……已连接  
# slave2 上的服务器：……已连接  
# 检查 server1 和 server2 上的 inventory 数据库  
  
WARNING: Objects in server1:inventory but not in server2:inventory:  
VIEW: finishing_up  
VIEW: cleaning
```

Type	Object Name	Defn Diff	Row Count	Data Check
TABLE	supplier	pass	FAIL	FAIL

Data differences found among rows:

```
--- inventory.supplier
```

```
+++ inventory.supplier
```

```
@@ -1,2 +1,2 @@
```

```
code,name
```

```
-2,Never Enough Inc.
```

```
+2,Wesayso Corporation
```

```
#
```

```
# 为 --changes-for=server1 生成转换语句
```

```
#
```

```
UPDATE `inventory`.`supplier` SET name = 'Wesayso Corporation' WHERE code = 2;
```

## 复制数据库：mysqldbcopy

624

有时候需要将数据从一台服务器复制到另一台服务器。根据需要复制的数据库的规模不同，以及复制的紧急程度和方便性，常规的离线方法，比如备份、文件复制或逻辑卷映像，都可能不方便搭建和执行。这个时候，简单地告诉让某个实用工具去复制数据库将会更容易。这就是复制数据库工具 *mysqldbcopy* 的任务，即复制每个数据库中的所有对象和数据。

复制数据库的工具能够指定从一台服务器复制一个或多个数据库到另一台服务器。在目标服务器上有几种方式重命名数据库。还可以在同一个服务器上指定一个不同的名字复制数据库，因此，不需要将原始数据库离线就可以重命名数据库。

### 用于复制

添加新 slave 时的其中一步是将来自 master 的数据装载到 slave，然后就可以在无强制 slave 从 master 读取并执行很多事件的情况下启动复制。虽然从 master（或另一个最新的 slave）恢复备份通常更有效更快，但是有时候少量地复制数据也是有必要或方便的。这个时候，*mysqldbcopy* 比部分恢复执行起来更简单。

### mysqldbcopy 示例

下面的例子从一台服务器向另一台服务器复制单个数据库。注意，复制执行的时候会输出信息。不仅对象和数据被复制，数据库级别的授权也会被复制。取决于备份应用程序

的情况，这个功能使得该工具比先复制数据再手动应用授权更快更简单。

```
$ mysqldbcoppy --source=root@slave1 --destination=root@slave2 \
  util_test:util_test
# slave1 上的 source: .....已连接
# slave2 上的 destination: .....已连接
# 复制 TABLE util_test.t1
# 复制表数据
# 复制 TABLE util_test.t2
# 复制表数据
# 复制 TABLE util_test.t3
# 复制表数据
# 复制 TABLE util_test.t4
# 复制表数据
# 复制 VIEW util_test.v1
# 复制 TRIGGER util_test.trg
# 复制 PROCEDURE util_test.pl
# 复制 UNCTION util_test.fl
# 复制 EVENT util_test.e1
# 复制 GRANTS from util_test
# .....完成。
```

625

## 导出数据库：mysqldbexport

使用 MySQL 的数据库复制工具复制数据包含两个操作，即从一台服务器读取数据和向另一台服务器写入数据。有时候，你想或者需要将数据的副本以文件形式保存，目的是为了操纵（手动更正多个行的错误），或安全保护，或导入分析工具。

为了实现这些使用场景，复制操作需要用到两个工具，即导出数据的 *mysqldbexport* 和导入数据的 *mysqldbimport*。导出和导入数据库的工具用于得到数据的一份逻辑副本。这意味着该工具一次一行地读取数据表，而不是像物理复制那样以二进制的形式从表中读取数据。

导出数据库工具使用 `--export` 选项复制对象的定义或数据，或者两者都复制。还可以通过 `--skip` 选项指定选择哪些对象类型。因此，可以选择只导出给定数据库的事件，或某个表的数据，或导出服务器上的所有对象和数据。

尽管这个工具在控制台（标准输出）上生成输出结果，你也可以将输出重定向到文件，以便其他工具使用，或由导入数据库工具使用。

这个工具的强大功能之一就是能够以多种格式生成输出。

### SQL

为创建对象（CREATE）和数据（INSERT）生成 SQL 语句。这是默认的输出格式。



## GRID

以网格或表的格式显示输出，像 mysql 客户端那样。

## CSV

以逗号分隔值的形式显示输出。

## TAB

以 tab 分隔的形式显示输出。

## VERTICAL

以单列的形式显示输出，像 mysql 客户端中的 \G 命令那样。



大量工具都支持这些格式，可以进一步过滤这些格式，让输出更易读或者更便于外部工具使用。

626

导出数据库工具能够通过向输出流添加适当的复制命令的方式支持复制。这样，你就能够将导出的结果作为 slave 或 master 的导入，这取决于你指定何种选项。使用 `--rpl` 选项告诉实用工具向输出结果添加 `STOP SLAVE`、`CHANGE MASTER`、`START SLAVE` 命令。例如，可以告诉实用工具使用复制命令以便在导入过程中控制 slave。为此，使用 `--rpl=SLAVE` 选项从 slave 复制数据，以更新另一个落后的 slave。`--rpl=SLAVE` 选项向输出的结尾添加一个 `CHANGE MASTER` 命令。当在另一个 slave 上导入这个文件的时候，这个 slave 就会连接到当时运行导出的那个 slave 所连接的 master。

`--rpl=MASTER` 选项也会向输出的结尾添加一个 `CHANGE MASTER` 命令，但是这次的 master 是运行导出的服务器。即，在另一个 slave 上导入这个文件的时候，它连接的是运行导出的那个 slave。

使用 `--rpl=BOTH` 选项能够同时生成上述两种语句，这样就可以在拓扑中按照不同的级别编辑导入的用途（当前服务器的 slave 及对等 slave）。

在生成复制语句的时候，它们总是以 SQL 语句的形式生成（无论选择哪种格式），并且按照导入操作时使用的顺序写。下面给出了生成语句的顺序（由 `-rpl` 选项生成的语句加粗表示）

1. **STOP SLAVE**
2. 对象定义
3. 数据

#### 4. CHANGE MASTER ...

#### 5. START SLAVE

工具还允许指定一个含有复制命令的独立文件，这样就可以从导出独立使用复制命令。在使用复制模式生成两种形式的 CHANGE MASTER（当前服务器作为 master 或者当前服务器的 master）的时候，这点非常有用。

回忆一下，CHANGE MASTER 命令需要复制用户名和密码。以 *用户名:密码* 的形式使用 --rpl-user 选项来指定。例如，--rpl-user=rpl:secret 选项告诉 *mysqldbexport* 使用 rpl 作为复制用户、secret 作为复制用户的密码，执行 CHANGE MASTER 命令：

```
CHANGE MASTER TO MASTER_HOST = 'localhost', MASTER_USER = 'rpl',
MASTER_PASSWORD = 'secret', MASTER_PORT = 13001,
MASTER_LOG_FILE = 'clone-bin.000001', MASTER_LOG_POS = 920;
```

**627** 如果省略了 --rpl-user 选项，CHANGE MASTER 命令会生成一个注释掉的部分，提示 master 的用户名和密码。因为在运行导出的时候这个信息可能还不可知。在稍后的例子中我们会看到这一点。

这个工具在复制中的另一个用途是，从数据损坏中恢复 slave，这也许是最有用的用途。与数据库一致性工具 *mysqldbcompare* 一起使用，*mysqldbexport* 能够从已知健康的 slave 中导出数据，然后在已恢复的 slave 上导入，以恢复一致性。

最新的 *mysqldbcopy* 和 *mysqldbexport* 都能够生成正确的 GTID 语句，设置 GTID\_PURGED 列表，用于在 slave 上导入数据。更多信息请参见在线参考手册的 MySQL 实用工具一节 (<http://bit.ly/mysql-utils>)。

### mysqldbexport 示例

下面的例子展示了如何使用复制语句和 GTID 命令导出数据库。这里使用的格式是逗号分隔的文件，不过复制命令都是 SQL 命令格式。注意，CHANGE MASTER 也是 SQL 命令格式。要使用这个命令，需要提供 master 上复制用户的用户名和密码。

虽然本例中的数据库很简单，不过这个例子阐明了在复制中使用这个工具所需的复制命令：

```
$ mysqldbexport --server=root:pass@localhost:13001 --export=both --rpl=master
--format=csv test_125
# localhost 上的 source: .....已连接
-- STOP SLAVE;
# 从 test_125 导出元数据
# test_125 中的 TABLES:
```

```

TABLE_SCHEMA, TABLE_NAME, ENGINE, ORDINAL_POSITION, COLUMN_NAME, COLUMN_TYPE,
IS_NULLABLE, COLUMN_DEFAULT, COLUMN_KEY, TABLE_COLLATION, CREATE_OPTIONS,
CONSTRAINT_NAME, REFERENCED_TABLE_NAME, UNIQUE_CONSTRAINT_NAME, UPDATE_RULE,
DELETE_RULE, CONSTRAINT_NAME, COL_NAME, REFERENCED_TABLE_SCHEMA,
REFERENCED_COLUMN_NAME
test_125, t1, InnoDB, 1, a, int(11), YES, , , latin1_swedish_ci, , , , , , , ,
test_125, t1, InnoDB, 2, b, char(30), YES, , , latin1_swedish_ci, , , , , , , ,
# test_125 中的 VIEWS : (未找到)
# test_125 中的 TRIGGERS : (未找到)
# test_125 中的 PROCEDURES : (未找到)
# test_125 中的 FUNCTIONS : (未找到)
# test_125 中的 EVENTS : (未找到)
# test_125 中的 GRANTS : (未找到)
# .....完成。
# 从 test_125 导出数据
# 表 test_125.t1 中的数据:
a,b
1,one
2,two
3,three
# .....完成。
# 以 master 方式连接当前服务器
# 警告: 未指定 --rpl-user, 找到多个具备复制权限的用户。
-- CHANGE MASTER TO MASTER_HOST = 'localhost', # MASTER_USER = '',
# MASTER_PASSWORD = '', MASTER_PORT = 13001,
MASTER_LOG_FILE = 'clone-bin.000001', MASTER_LOG_POS = 920;
-- START SLAVE;

```



注意，以 -- 开头的表示复制命令。*mysqldbimport* 能够使用这些复制命令导入文件并识别数据行。

## 导入数据库：mysqldbimport

导入数据库工具 *mysqldbimport* 以导出数据库工具所生成的格式读入文件。因此，我们能够以最方便的格式恢复或使用数据的副本，然后在无须格式转换的情况下导入数据。

导入数据库工具简单地读取文件，解析那些非 SQL 格式并将它们转换为 SQL 语句，然后在服务器上执行这些命令。



如果你的文件格式是 *mysqldbexport* 的输出格式，就可以使用 *mysqldbimport* 导入文件。

这个工具的最佳功能之一是仅导入对象定义或数据。使用 `--import` 选项告诉实用工具导入对象定义，或数据，或两者都导入。

另一个功能是在导入的时候更改数据表的存储引擎，从而能够一次完成两个操作。如果导入数据的服务器不支持原始服务器的存储引擎，这个功能就很方便。例如，用这个功能一步完成将存储引擎更改为 InnoDB，以恢复 MyISAM 表。

最后，可以使用 `--dryrun` 选项验证文件，而并不是真的执行那些生成的语句。

## 用于复制

*mysqldbimport* 同 *mysqldbexport* 配对使用，可支持复制用例。导入操作通常紧随导出之后，即从一个服务器读取然后写入另一个服务器。如果要导入的文件中含有复制命令，则在遇到这些命令的时候执行它们。

## mysqldbimport 示例

下面的例子导入前面“*mysqldbexport* 示例”一节中的数据库导出的数据。这里，我们告诉导入数据库工具同时导入对象定义和数据：

```
$ mysqldbimport --server=root:pass@localhost:13001 test_125.sql --import=both
# localhost 上的 source：……已连接。
# 从 test_125.sql 导入定义和数据。
# ……完成。
```

当使用 *mysqldbimport* 工具导入 *mysqldbexport* 工具的输出时，如果 *mysqldbexport* 的输出中含有复制命令（通过 `--rpl` 命令），只需要执行这些命令即可。*mysqldbimport* 工具能够识别并执行这些命令，即使文件格式不是 SQL 格式。

不过，你也可以使用 `--skip-rpl` 选项禁用这些命令。如果导出结果被保存到文件，而你导入的服务器并不是或不打算成为拓扑的一部分，那么会用到这个选项。例如，如果想要生成一个测试服务器来测试某个应用程序，可以使用某个可用 slave 的导出结果，并在测试服务器上执行导入的时候跳过复制命令。

## 发现不同：mysqldiff

数据库差异工具 *mysqldiff* 执行对象定义差异检查，这在 *mysqldbcompare* 工具中用到过，



前面已经介绍过 *mysqldbcompare* 工具。实际上, *mysqldiff* 比 *mysqldbcompare* 出现得更早, 并且包含 *mysqldbcompare* 的全部功能。所以 MySQL 实用工具支持库存在很大程度的重用。

数据库差异工具并不能替代数据一致性工具, 而是仅在用户需要检查对象定义的时候使用这个工具。

*mysqldiff* 的输出是一个差异报告, 格式由你自己决定, 包括统一格式 (unified)、上下文格式 (context)、差异格式 (differ) 或 SQL 格式 (SQL)。默认为统一格式。SQL 格式很有用, 因为它会生成转换语句 (ALTER 语句), 用于转换数据表, 保持表结构一致。

◀ 630

## 用于复制

*mysqldiff* 能够帮助复制检测什么时候对象出现分歧。例如, 某个服务器更新数据失败, 是因为一个或多个表或者相互依赖的表的结构发生了改变。



MySQL 复制允许 master 和 slave 上使用不同的字段名。这个功能有很多合法用例。但是, 目前 *mysqldiff* 无法识别差异是否合法, 而是将这些用例作为错误识别出来。

## mysqldiff 示例

下面的例子使用这个 *mysqldiff* 在两个服务器上检测对象的差异:

```
$ mysqldiff --server1=root:pass@localhost:13001
           --server2=root:pass@localhost:13002
```

```
diff_db:diff_db
```

```
# localhost 上的 server1: .....已连接。
```

```
# localhost 上的 server2: .....已连接。
```

```
# 比较 diff_db 和 diff_db
```

[ 通过 ]

```
# 比较 diff_db.table1 和 diff_db.table1
```

[ 失败 ]

```
# 不一致的对象定义 (--changes-for=server1)
```

```
#
```

```
--- diff_db.table1
```

```
+++ diff_db.table1
```

```
@@ -1,4 +1,5 @@
```

```
CREATE TABLE `table1` (
```

```
  `a` int(11) DEFAULT NULL,
```

```
- `b` char(30) DEFAULT NULL
```

```
+ `b` char(30) DEFAULT NULL,
```

```
+ `c` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
```

) ENGINE=InnoDB DEFAULT CHARSET=latin1  
Compare failed. One or more differences found.

## 检测错误的写操作

复制出现错误的其中一个途径是，数据更新被错误地定向到 slave，而不是 master。如果 slave 不是以只读模式运行的，用户就可能会在错误的服务器上发出更新操作。如果出现这种情况，某段时间内可能检测不到，从而带来一些破坏性的副作用。

可能表现为：对于那些在其他 slave 上正确执行的查询，slave 突然停止，或者应用程序在某些 slave 上的数据显示与其他 slave 上的不同。

有一个技巧有助于识别这些错误的更新。如果表对时间戳没有任何限制，向 slave 上的表添加时间戳字段。默认情况下，任何复制事件都不会改变时间戳字段的值，而 slave 上的数据变更则会改变字段值。

下面的代码展示了如何进行设置，并识别那些发生变化的行（我们对代码进行了注释，为了简洁性，还去掉了其他无关输出；这些命令是在由一个 master 和一个 slave 构成的简单复制拓扑上运行的）：

```
# 在 master 上运行
mysql> CREATE DATABASE diff_db;
mysql> CREATE TABLE diff_db.table1 (a int, b char(30));
mysql> INSERT INTO diff_db.table1 VALUES (1, 'one'), (2, 'two'), (3, 'three');
mysql> SELECT * FROM diff_db.table1;
+-----+-----+
| a     | b     |
+-----+-----+
| 1     | one   |
| 2     | two   |
| 3     | three |
+-----+-----+

# 在 slave 上运行
mysql> SELECT * FROM diff_db.table1;
+-----+-----+
| a     | b     |
+-----+-----+
| 1     | one   |
| 2     | two   |
| 3     | three |
+-----+-----+
```

# 在 slave 上更改表：

```
mysql> ALTER TABLE diff_db.table1
add column c timestamp default NOW() after b;
```

### 输入错误的更新 ###

```
mysql> INSERT INTO diff_db.table1 VALUES (4, 'four', NULL);
```

```
mysql> SELECT * FROM diff_db.table1;
```

a	b	c
1	one	0000-00-00 00:00:00
2	two	0000-00-00 00:00:00
3	three	0000-00-00 00:00:00
4	four	2012-11-14 15:34:08

# 在 master 上运行

```
mysql> INSERT INTO diff_db.table1 VALUES (7, 'seven'), (8, 'eight');
```

```
mysql> SELECT * FROM diff_db.table1;
```

a	b
1	one
2	two
3	three
7	seven
8	eight

# 在 slave 上运行

```
mysql> SELECT * FROM diff_db.table1;
```

a	b	c
1	one	0000-00-00 00:00:00
2	two	0000-00-00 00:00:00
3	three	0000-00-00 00:00:00
4	four	2012-11-14 15:34:08
7	seven	0000-00-00 00:00:00
8	eight	0000-00-00 00:00:00

```
# 检测错误更新的查询
```

```
mysql> SELECT * FROM diff_db.table1 WHERE c > 0;
```

```
+-----+-----+-----+
| a      | b      | c      |
+-----+-----+-----+
|      4 | four   | 2012-11-14 15:34:08 |
+-----+-----+-----+
```

注意，最后两个查询是在 slave 上发出的。我们看到，来自 master 的行的时间戳字段并没有改变（为了验证这一点，我们显示了添加的新行），而 slave 上更新的行则得到了新的时间戳值。如果你不想让你的 slave 只读，那么可以考虑类似的解决方法。

## 显示磁盘使用情况：mysqldiskusage

磁盘使用情况工具 *mysqldiskusage* 是一种诊断工具，提供关于 MySQL 服务器的信息报告。这个工具显示每个数据库的总规模以及其他文件相关信息，比如日志文件名和大小，以及 InnoDB 配置参数。

**633** 数据库规模有两种计算方法。如果用户账号对数据目录具有读权限，那么数据库规模反映了该目录下文件的总大小。如果用户不能访问数据目录，则根据表的元数据信息计算数据库规模，包括行的大小和行的平均长度。默认情况下，该工具只显示数据库规模，不过，`--verbose` 选项能够分别显示数据和各种文件的大小。

使用 `--binlog` 选项，还可以显示已有的二进制日志及其数目，使用 `--relaylog` 选项可显示中继日志，使用 `--logs` 可显示慢查询日志，使用 `--all` 选项可开启所有选项。

输出格式为 GRID、CSV、TAB 或 VERTICAL，这在前面 *mysqldbexport* 工具一节中介绍过。



这个工具必须在装有 MySQL 的服务器上运行。需要具备数据目录的访问权限才能显示关于日志和 InnoDB 数据的全部信息。所以，运行这个工具可能需要提升权限。

### 用于复制

虽然 *mysqldiskusage* 不会直接对复制起作用，但是在计划升级复制服务器的时候有助于确定你是否拥有足够的磁盘空间。此外，还可以使用这个工具检查日志文件和 InnoDB 文件的大小。



## mysqldiskusage 示例

下面的例子详细列出了某个服务器上的所有数据库的规模。我们还启用了所有选项，显示了所有信息。注意，这里显示的是 InnoDB 的信息。这有助于诊断 InnoDB 的配置问题。

```
$ mysqldiskusage --server=root:root@localhost --binlog --logs --innodb --all
--empty -vvv
```

# localhost 上的 source: .....已连接。#所有数据库:

db_name	db_dir_size	data_size	misc_files	total
oltp2	472,785	445,728	27,057	472,785
frm_test	472,785	445,728	27,057	472,785
compare_util_test	472,785	445,728	27,057	472,785
bvm	15,129	6,080	9,049	15,129
compare	97,924,346	97,897,677	26,669	97,924,346
compare_employees	554,176,356	554,166,900	9,456	554,176,356
db1	11,035	2,188	8,847	11,035
db2	11,035	2,188	8,847	11,035
emp1	17,517	81,920	17,517	99,437
emp2	88,877	206,029,917	87,760	206,117,677
emp3	27,989	33,885	26,872	60,757
employees	88,892	206,029,917	87,775	206,117,692
griots	14,415	5,288	9,127	14,415
groits	9,045	16,384	9,045	25,429
mysql	903,022	693,480	191,720	885,200
oltp1	62,705	114,688	62,705	177,393
sak1	411,538	6,887,232	143,442	7,030,674
sakila	510,111	6,944,576	184,671	7,129,247
test	580,636	74,029	506,607	580,636
test123	17,343	32,768	17,343	50,111
test_1	48,730	5,225	43,505	48,730
test_cache	9,868	1,045	8,823	9,868
test_trig	9,871	1,024	8,847	9,871
util_test_clone2	36,709	50,269	35,592	85,861
welford_kindle	59,058	48,896	10,162	59,058
world	472,785	445,728	27,057	472,785

634

Total database disk usage = 1,081,112,742 bytes or 1.00 GB

# 日志信息:

log_name	size
Chucks-iMac.log	829,601,162

Chucks-iMac-slow.log	97,783
Chucks-iMac.local.err	234,028

Total size of logs = 829,932,973 bytes or 791.00 MB

# 二进制日志信息:

Current binary log file = my\_log.000171

log_file	size
my_log.000001	3527
my_log.000002	785115
my_log.000003	569
...	
my_log.000169	348
my_log.000170	258
my_log.000171	722
my_log.index	2736

Total size of binary logs = 21,883,426 bytes or 20.00 MB

# 服务器不是一个活动的 slave, 所以没有中继日志信息。

# InnoDB 表空间信息:

innodb_file	size	type	specification
ib_logfile0	5242880	log file	
ib_logfile1	5242880	log file	
ibdata1	815792128	shared tablespace	ibdata1:778M
ibdata2	52428800	shared tablespace	ibdata2:50M:autoextend

Total size of InnoDB files = 889,192,448 bytes or 848.00 MB

Tablespace ibdata2:50M:autoextend can be extended by using ibdata2:50M[...]

InnoDB freespace = 427,819,008 bytes or 408.00 MB

# .....完成。

注意二进制日志信息。为了简洁, 我们去掉了部分结果, 不过从这段代码可清楚地看到, 服务器从未清除(purge)二进制日志。如果服务器已经运行了很长时间, 这可能需要注意。

## 检查表的索引：mysqlindexcheck

为了改善查询最需要检查的地方是索引。索引检查工具 *mysqlindexcheck* 对表进行检查，以确定是否存在重复或冗余的索引。重复的索引是指那些完全一样的索引，而冗余的索引是指某些索引是其他索引的子集。例如，如果 *id* 和 *title* 字段上有个索引（就是这个顺序），而 *id* 字段也有一个索引，那么这个单列上的索引就是不需要的。你会惊讶地发现这种情况是多么常见。

*mysqlindexcheck* 能够为所有检测到的索引生成 DROP 命令的 SQL 语句。还可以显示这些识别到的索引的统计数据。这有助于某些索引参数的性能调优。

也许最有用的选项是显示最差（或最佳）性能的索引。这在改善数据库的查询性能时非常有用。

### 用于复制

虽然这个工具并不直接对复制起作用，但却能够提升查询的性能，因此使用这个工具能够帮助你提升那些运行在复制系统中的应用程序的性能。

### mysqlindexcheck 示例

下面的例子展示了在样本数据库 Sakila 上检查重复和冗余索引的结果（由于未找到重复或冗余索引，我们使用 *--worst* 选项显示了每个表的最差性能索引；为了简洁，省略了部分数据）：

```
$ mysqlindexcheck --stats -d -s --server=root:pass@localhost sakila --worst=1  
--format=vertical
```

636

```
# localhost 上的 source：……已连接。
```

```
#
```

```
# 显示 sakila.actor 的第 1 个最差性能索引：
```

```
#
```

```
***** 1. row *****
```

```
database: sakila
```

```
table: actor
```

```
name: idx_actor_last_name
```

```
column: last_name
```

```
sequence: 1
```

```
num columns: 1
```

```
cardinality: 200
```

```
est. rows: 200
```

```
percent: 100.00
```

```
1 rows.
```

```

.....
#
# 显示 sakila.rental 的第 1 个最差性能索引：
#
***** 1. row *****
    database: sakila
      table: rental
      name: idx_fk_staff_id
      column: staff_id
      sequence: 1
    num columns: 1
    cardinality: 1
      est. rows: 16305
      percent: 0.01
1 rows.

```

## 查找元数据：mysqlmetagrep

元数据查找工具 *mysqlmetagrep* 是另一个诊断工具，可帮助发现匹配给定模式的对象。任何需要定位匹配某个名称或部分名称的对象的时候，这个工具都非常有用。

这个实用工具具备独有的查找能力，允许使用 POSIX 正则表达式（REGEX）或 SQL 模式（如 LIKE）进行查找。使用 `--object-types` 选项限制查找特定的对象类型，该选项接受以逗号分隔的参数列表，包括 `database`、`trigger`、`user`、`routine`、`column`、`table`、`partition`、`event` 和 `view`。默认查找所有的对象类型。虽然默认按名称查找，但也可以使用 `--body` 选项告诉实用工具搜索存储过程的主体。

还可以为查找生成一个 SQL 语句，以便在其他例程或脚本中使用。

**637** 输出的格式为 GRID、CSV、TAB 或 VERTICAL，这在 *mysqldbexport* 部分介绍过。

### 用于复制

如果你遇到应用程序错误，或者导致不完整语句的二进制日志，或者不完整信息的错误和警告，就需要按名称查找对象。

### mysqlmetagrep 示例

下面的例子查找以字母 t 开头的对象的元数据（我们通过 t 后面加上下画线符号来表示）：

```

$ mysqlmetagrep --server=root:pass@localhost --pattern="t_" --format=vertical
***** 1. row *****

```



```

Connection: root:*@localhost:3306
Object Type: TABLE
Object Name: t1
Database: example1
Field Type: TABLE
Matches: t1
*****
2. row *****
Connection: root:*@localhost:3306
Object Type: TABLE
Object Name: t1
Database: test
Field Type: TABLE
Matches: t1
2 rows.

```

## 查找进程：mysqlprocgrep

进程查找工具 *mysqlprocgrep* 使用与元数据查找工具一样的查找功能，对服务器上的运行进程进行查找。这时，可以匹配 SHOW PROCESSLIST 输出结果中的 user、host、database、command、info 和 state 字段。还可以按照 age 查询进程，从而识别长时间运行的查询或短时连接。一旦发现感兴趣的进程，就可以杀死匹配查找项的查询或连接。



使用查找项运行实用工具，直到找到你想要的那些进程。然后再次运行工具，指定 --kill-query 或 --kill-connection 选项，来杀死匹配进程的查询或连接。

与元数据查找工具类似，可以使用 --sql 选项生成 SQL 语句，以便在其他存储过程或脚本中使用。

638

### 用于复制

进程查找工具有助于在复制中识别每个 slave 的连接，或者按照用户名或主机名识别某个 slave 的连接。我们将在示例中使用这个工具。

### mysqlprocgrep 示例

下面的例子展示了某个复制工具生成的拓扑图，这个工具我们稍后介绍（参见本章后面“显示拓扑结构：mysqlrplshow”小节）。在这个例子中，有一个 slave 没有使用 --report-host 或 --report-port 选项启动（所有复制工具都要求使用这些选项）。首先

我们展示拓扑图，其中包括一个 master 和三个 slave。注意警告信息，这个工具检测到不止三个 slave。使用进程查找工具能够发现第四个 slave。

```
$ mysqlrplshow --master=root:pass@localhost:13002 --disco=root:pass
```

```
# localhost 上的 master: .....已连接。
```

```
# 为 master 寻找 slave: localhost:13002 警告: 存在没有使用 --report-host 或 --report-port 注册的 slave
```

```
# 复制拓扑图
```

```
localhost:13002 (MASTER)
```

```
|
+--- slavehost1:13001 - (SLAVE)
|
+--- slavehost3:13003 - (SLAVE)
|
+--- slavehost4:13004 - (SLAVE)
```

```
$ mysqlprocgrep --server=root:pass@localhost:13002 --match-command="Bin%"
--format=vertical
```

```
***** 1. row *****
```

```
Connection: root:*@localhost:13002
```

```
Id: 10
```

```
User: rpl
```

```
Host: slavehost1:56901
```

```
Db: None
```

```
Command: Binlog Dump
```

```
Time: 258
```

```
State: Master has sent all binlog to slave; waiting for binlog to be up
```

```
Info: None
```

```
***** 2. row *****
```

```
Connection: root:*@localhost:13002
```

```
Id: 8
```

```
User: rpl
```

```
Host: slavehost3:56898
```

```
Db: None
```

```
Command: Binlog Dump
```

```
Time: 261
```

```
State: Master has sent all binlog to slave; waiting for binlog to be up
```

```
Info: None
```

```
***** 3. row *****
```

```
Connection: root:*@localhost:13002
```

```
Id: 6
```

```
User: rpl
```

```
Host: slavehost4:56894
```

```
Db: None
```

```
Command: Binlog Dump
```

639

```

Time: 264
State: Master has sent all binlog to slave; waiting for binlog to be up
Info: None
*****
4. row *****
Connection: root:*@localhost:13002
Id: 19
User: rpl
Host: localhost:56942
Db: None
Command: Binlog Dump
Time: 18
State: Master has sent all binlog to slave; waiting for binlog to be up
Info: None
4 rows.

```

## 克隆服务器：mysqlserverclone

服务器克隆工具 *mysqlserverclone* 是一个主力工具。它能够创建一个服务器的新实例，方法是连接到正在运行的服务器，或者从 MySQL 启动一个服务器（使用 `--basedir` 选项）。

使用这个工具的时候，需要指定一个新的数据目录和新端口。还要指定服务器 ID 和 root 用户的密码，然后传递任何特定的选项，启动 *mysqld* 进程。如果使用 `--verbose` 选项，工具就会在启动的时候打印服务器消息；否则不打印。

新服务器启动的时候只有一个默认的数据库。例如，你只能看到 *mysql* 数据库（以及 *INFORMATION\_SCHEMA* 和 *PERFORMANCE\_SCHEMA*）。如果想向新服务器复制数据，可以使用数据库复制或者数据库导出导入工具。

### 用于复制

当需要快速创建新 slave 的时候，这个工具非常方便。在大多数系统上，这个工具只需要几秒钟就可以创建新的数据目录并启动服务器。这样，快速建立复制 slave 就很容易，甚至可以在单个服务器上建立整个拓扑。

如果你面临二进制日志的诊断难题，或复制错误，或数据一致性问题，可以使用 *mysqlserverclone* 快速创建并测试 master 和 slave。然后，通过复制服务器实例并在这些实例上重建错误，对问题进行诊断调查，这样就不会对已有服务器造成损害。

640

### mysqlserverclone 示例

下面的例子展示了如何为一个可能离线或不可达的服务器启动一个新实例。在安装 MySQL 的主机上运行这个命令。传递的参数包括新的数据目录、新的端口、新的服务

器 ID 和 root 密码：

```
$ mysqlserverclone --basedir=/usr/local/mysql-5.6 --new-data=../temp_13001 \  
--new-port=13001 --new-id=101 --root=pass123 \  
--mysqld="--log-bin --gtid-mode=on --log-slave-updates  
--enforce-gtid-consistency"
```

# 在 ../mysql-5.6 克隆 MySQL 服务器。

# 创建新的数据目录……

# 配置新实例……

# 定位 mysql 工具……

# 设置空数据库和 mysql 表……

# 启动服务器的新实例……

# 测试新实例的连接……

# 成功!

# 设置 root 密码……

# 连接信息：

# -uroot -ppass123 --socket=../temp\_13001/mysql.sock

# ……完成。

下面的例子展示了在一个运行着的服务器上启动新实例是多么简单。注意，使用 `--mysqld` 选项向服务器进程（mysqld）传递的参数。还要注意我们使用了 `--delete-data` 选项，如果数据目录存在则删除它。只有当数据目录为空或者使用了该选项的时候，这个工具才能启动：

```
$ mysqlserverclone --server=root:root@localhost:13001 --new-data=../temp_13002 \  
--new-port=13002 --new-id=102 --root=pass123 \  
--mysqld="--log-bin --gtid-mode=on --log-slave-updates  
--enforce-gtid-consistency" \  
--delete-data
```

# 克隆正在 localhost 上运行的 MySQL 服务器。

# 创建新的数据目录……

# 配置新实例……

# 定位 mysql 工具……

# 设置空数据库和 mysql 表……

# 启动服务器的新实例……

# 测试新实例的连接……

# 成功!

# 设置 root 密码……

# 连接信息：

# -uroot -ppass123 --socket=/Volumes/Source/source/temp\_13002/mysql.sock

# ……完成。

641



## 显示服务器信息：mysqlserverinfo

显示服务器信息命令 *mysqlserverinfo* 也是一个诊断工具，提供关于服务器的基本信息，显示服务器的主要配置，包括基本目录、数据目录、配置文件及更多信息。

有几个选项用于显示配置文件的默认信息，还有一个选项能够显示运行该工具的主机上正在运行的 MySQL 服务器列表。这个命令能够以多种方式显示信息，包括 GRID、CSV、TAB 和 VERTICAL，这在本章前面“导出数据库：mysqldbexport”一节中介绍过。

### 用于复制

正如任何诊断工具一样，这个工具适用于复制中所有类型的问题调查。如果同服务器克隆工具一起使用，能够快速发现任何克隆的服务器实例。

### mysqlserverinfo 示例

下面的例子在 localhost 上的服务器上运行这个工具。其中使用的选项列出了正在这个主机上运行的服务器，以及配置文件的默认信息：

```
$ mysqlserverinfo --server=root:root@localhost:13001 --show-servers
--show-defaults --format=VERTICAL
```

```
#
```

```
# 这个主机上活动的 MySQL 服务器如下：
```

```
# Process id: 236, Data path: /Users/cbell/source/temp_13001
```

```
# Process id: 331, Data path: /Users/cbell/source/temp_13002
```

```
# Process id: 426, Data path: /Users/cbell/source/temp_13003
```

```
# Process id: 431, Data path: /Users/cbell/source/temp_13004
```

```
#
```

```
# localhost 上的 source：……已连接。
```

```
***** 1. row *****
```

```
server: localhost:13001
```

```
version: 5.6.6-m9-log
```

```
datadir: /Users/cbell/source/temp_13001/
```

```
basedir: /Users/cbell/source/mysql-5.6/
```

```
plugin_dir: /Users/cbell/source/mysql-5.6/lib/plugin/
```

```
config_file: /etc/my.cnf
```

```
binary_log: clone-bin.000001
```

```
binary_log_pos: 853
```

```
relay_log: None
```

```
relay_log_pos: None
```

```
1 rows.
```

```
Defaults for server localhost:13001
```

```
--port=13001
```

```
--basedir=/usr/local/mysql
--datadir=/usr/local/mysql/data
--server_id=5
--log-bin=my_log
--general_log
--slow_query_log
--innodb_data_file_path=ibdata1:778M;ibdata2:50M:autoextend
# .....完成
```



使用 `--show-defaults` 选项的时候必须在 `localhost`（服务器）上运行该工具。

## 克隆用户：mysqluserclone

MySQL 的数据库管理员经常抱怨创建具备同样权限的多个用户很不方便。机智的管理员常常会编写脚本来实现这一点，通过 `GRANT` 语句，为每个需要创建的用户提供用户名和主机。

这种做法很容易出错，特别是在权限随时间发生变化的时候。如果安全性是一件高优先级的事情，那么这个脚本的位置和访问权限需要特别注意。

看上去这需要大量的额外工作，幸好，有一个工具能够使复制（克隆）用户权限变得很容易。用户克隆工具 *mysqluserclone* 将一个用户的所有权限复制到另一个或多个用户，这些用户可以在同一个系统中或者处于不同的系统，甚至可以为每个创建的用户设置密码。

这个工具有很多有趣的功能，比如在克隆操作过程中使用 `SQL` 语句，以及使用匹配 *user@host* 指定的全局权限。使用 `--list` 选项可以列出全部用户及其权限。输出格式为 `GRID`、`CSV`、`TAB` 和 `VERTICAL`，这在本章前面“输出数据库：*mysqldbexport*”一节中介绍过。

### 用于复制

该工具在 `slave` 的提供（*provision*）过程中很有用。这时，你有一个新 `slave`，需要重新创建那些 `master` 上已有的账号。用户克隆工具简单地指定想要克隆的 `master`（或另一个活动 `slave`）上的用户，并提供 `master` 和新 `slave` 的连接信息。甚至可以使用一个选项克隆所有用户。因此，这个工具是为简化 `slave` 提供的又一个方便的工具。

#### 643 mysqluserclone 示例

下面的例子从一台服务器向另一台服务器克隆用户账号及其权限。第一个命令输出（转

储) 这个用户的 GRANT 语句。第二个命令执行克隆操作。注意, 我们在第二个命令中指定了两次用户。这是因为该工具允许从单个用户账号克隆其他用户。所以在这个命令中, 第一个实例是你想要复制的用户, 第二个实例是你想要创建的具备同样权限的新用户。可以随便指定多少个新用户, 每一个都是对已有用户的克隆:

```
$ mysqluserclone --source=root:root@localhost:13002 joe@localhost --dump
# localhost 上的 source: .....已连接。
Dumping grants for user joe@localhost
GRANT SELECT ON *.* TO 'joe'@'localhost'

$ mysqluserclone --source=root:root@localhost:13002
--destination=root:root@localhost:13003 joe@localhost joe@localhost
# localhost 上的 source: .....已连接。
# localhost 上的 destination: .....已连接。
# 克隆一个用户.....
# 将 joe@localhost 克隆到 joe@localhost
# .....完成。
```

## 实用工具客户端: mysqluc

MySQL 实用工具客户端 *mysqluc* 本身并不是一个实用工具, 而是一个为实用工具命令、选项和参数提供代码补全的包装器。它可以指定用户自定义的变量, 从而重复使用单个值。

也许实用工具客户端的最佳功能是辅助任何实用工具, 包括它自己。这就使得任何实用工具的使用变得更简单更方便。

### 用于复制

客户端并不提供任何与复制直接相关的管理性功能, 但却是一个运行复制工具的便利工具。特别是在使用了用户自定义变量表示 master 和 slave 连接的情况下。代码补全还能帮助你记住各种选项, 对那些不常使用的选项特别有用。

### mysqluc 示例

下面的例子在实用工具控制台内部执行了若干命令。这些命令展示了如何显示一个简单复制拓扑的健康状况。稍后我们会进一步讨论复制管理工具 (*mysqlrpladmin*):

```
$ mysqluc
Launching console ...
Welcome to the MySQL Utilities Client (mysqluc) version 1.1.0 -
MySQL Workbench Distribution 5.2.44
Copyright (c) 2000, 2012, Oracle and/or its affiliates. All rights reserved.
```

Oracle is a registered trademark of Oracle Corporation and/or its affiliates.  
Other names may be trademarks of their respective owners.

Type 'help' for a list of commands or press TAB twice for list of utilities.

```
mysqluc> help rpladmin
```

```
Usage: mysqlrpladmin.py --slaves=root@localhost:3306 <command>
```

```
mysqluc> set $MASTER=root:root@localhost:13001
```

```
mysqluc> set $DISCOVER=root:root
```

```
mysqluc> rpladmin --master=$MASTER --disco=$DISCOVER health
```

```
# 为 localhost:13001 上的 master 查找 slave
```

```
# 检查权限
```

```
#
```

```
# 复制拓扑健康状况：
```

host	port	role	state	gtid_mode	health
localhost	13001	MASTER	UP	ON	OK
localhost	13002	SLAVE	UP	ON	OK
localhost	13003	SLAVE	UP	ON	OK
localhost	13004	SLAVE	UP	ON	OK

```
# .....完成
```

注意，这些命令都是在客户端发出的。客户端还有一个很方便的功能，即你不需要在实用工具命令中键入 *mysql* 前缀。还要注意用户自定义变量的使用，即 *\$MASTER* 和 *\$DISCOVER*，以及复制管理工具的帮助内容。显然，这个工具非常方便。

## 复制的实用工具

这一节简单介绍那些专门用于复制环境的 MySQL 实用工具。

### 配置复制：mysqlreplicate

在第 3 章中，我们已经详细讨论过复制的配置。总的来说，只要满足复制的前提条件，按照以下步骤进行操作即可：

1. 在 master 上创建复制用户。

645 2. 如果不使用 GTID，检查 master 的 SHOW MASTER STATUS 视图，并记录这些值。



3. 在 slave 上发出 `CHANGE MASTER TO` 命令。
4. 在 slave 上发出 `START SLAVE` 命令。
5. 在 slave 上使用 `SHOW SLAVE STATUS` 检查错误。

虽然这些操作并不多，但至少需要访问一次 master，master 至少需要发出两个命令，slave 至少需要发出三个命令。如果在大型横向扩展时这样操作很多次的话，就变得很冗繁。

这个时候，复制实用工具 *mysqlreplicate* 能够让事情变得简单。它不需要在 master 及其 slave 上发出 5 个命令，复制命令只需使用单个命令列出 master 和 slave 即可，然后由工具完成剩下的工作。有若干选项可以设置复制用户、测试复制配置，最重要的是能够控制复制从哪里开始。

复制可以从起点开始（即 slave 向 master 请求所有事件），或从某个二进制日志文件的第一个事件开始，或者从某个二进制日志文件和位置开始。这个工具提供了足够的选项涵盖了复制从哪里开始的场景。

最棒的是，由于只使用单个命令，所以你很容易编写脚本，以用于大型横向扩展操作，只需简单地使用一个变量即可，该变量的值为一组 slave（甚至 master）。

## mysqlreplicate 示例

下面的例子展示了使用默认选项的简单复制配置：

```
$ mysqlreplicate --master=root:root@localhost:13001
--slave=root:root@localhost:13002
# localhost 上的 master: .....已连接。
# localhost 上的 slave: .....已连接。
# 检查 master 上的二进制日志.....
# 配置复制.....
# .....完成。
```

下面的例子使用 `-vvv` 选项执行上面的命令可获取最详细的信息。注意，复制配置所需的全部命令都由实用工具完成，包括重要的先决条件检查。如果使用这个工具发现复制配置出错，那么使用详情选项重新运行这个工具，检查错误的输出信息：

```
$ mysqlreplicate --master=root:root@localhost:13001
--slave=root:root@localhost:13002 -vvv --test-db=test_rpl
# localhost 上的 master: .....已连接。
# localhost 上的 slave: .....已连接。
# master id = 101
```

```

# slave id = 102
# master uuid = 8e1903fa-2de4-11e2-a8c9-59dc46c8181e
# slave uuid = 8a40ec3a-2ded-11e2-a903-0ca829d3a6c5
# 检查 InnoDB 统计信息的类型和版本冲突。
# 检查存储引擎……
# 检查 master 上的二进制日志……
# 配置复制……
# 将 slave 连接到 master……
# CHANGE MASTER TO MASTER_HOST = 'localhost', MASTER_USER = 'rpl',
MASTER_PASSWORD = 'rpl', MASTER_PORT = 13001, MASTER_AUTO_POSITION=1
# 从 master 的最后位置启动 slave……
# 状态：等待 master 发送事件
# 错误：0：
# 测试复制配置……
# 在 master 上创建一个名为 test_rpl 的测试数据库……
# 成功！复制正在运行。
# ……完成。

```

注意，我们还使用了 `--test-db` 选项来测试 master 和 slave 之间的复制。指定了这个选项后，工具会创建一个新的数据库，并保证 slave 能够接收更新。因此，工具能够在 slave 启动以后确保复制正常运行。

对于配置复制拓扑来说，这个工具是最有用的复制工具。但是，如果你还不知道 slave 的完整配置，或者不知道 master 和 slave 的数量，那么检查配置是否正确，或者确定服务器是否能作为 master 和 slave 使用，是一件令人厌烦的事情。幸运的是，下面这个工具能够满足这个需要。

## 检查复制的配置：mysqlrplcheck

设置复制的挑战之一就是确定 master 和 slave 什么时候具备了全部的先决条件，并且是兼容的。这里，“兼容”是指 slave 的设置正确，它要么重新加入 master，要么第一次连接 master。

复制检查工具 *mysqlrplcheck* 检查 master 和 slave，确定它们每一个的复制配置是否都正确（例如 master 启用了二进制日志），是否兼容，以及 slave 是否曾经连接过或正在连接 master。如果你不确定 slave 应该连接哪个 master，特别是 slave 刚从错误条件恢复的时候，那么最后这个功能就很方便。

由于这个工具处理一个 master 和一个 slave，所以很适合用于复制中的诊断和管理性工作。如果使用 `-vvv` 级别的详细度运行这个工具，就会看到很多额外的详细信息，包括 UUID 的真实值、slave 的状态，以及更多。详细信息显示还提供了更多关于错误条件的信息。

## mysqlrplcheck 示例

我们来看一个例子，这里有一个 master，并且 slave 没有连接到 master，存在配置冲突。下面的输出结果展示了 master 和 slave 之间的很多复制问题（我们使用详细信息选项查看额外详情）：

```
$ mysqlrplcheck --master=root:root@localhost:13001
--slave=root:root@localhost:13002 -vvv
# localhost 上的 master: .....已连接。
# localhost 上的 slave: .....已连接。
Test Description                                     Status
-----
Checking for binary logging on master                [pass]
Are there binlog exceptions?                         [pass]
Replication user exists?                             [FAIL]

Slave is not connected to a master.

Checking server_id values                             [FAIL]

The slave's server_id is the same as the master.

Checking server_uuid values                           [pass]

master uuid = 8e1903fa-2de4-11e2-a8c9-59dc46c8181e
slave uuid = d722e59c-2de5-11e2-a8d1-3f758e665783

Is slave connected to master?                        [FAIL]

Slave is stopped.

Check master information file                         [pass]

#
# master 信息文件：
#
# 从表中读取 master 信息。

Checking InnoDB compatibility                        [pass]
Checking storage engines compatibility                [pass]
Checking lower_case_table_names settings             [pass]
```

```
Master lower_case_table_names: 2
Slave lower_case_table_names: 2
```

```
Checking slave delay (seconds behind master)
```

```
[FAIL]
```

648 The server specified as the slave is not configured as a replication slave.

```
# .....完成。
```

注意其中揭示的问题。由于 slave 没有连接 master，很多检查或测试都失败了，比如复制使用检查，slave 连接检查和 slave 中继检查。不过最重要的是，我们看到 master 和 slave 上的服务器 ID 一样，这是导致无法配置复制的一个常见问题。



如果 slave 已经连接上 master，并且还有错误，详细信息选项可以显示 SHOW SLAVE STATUS 视图的输出结果。

下面的代码是成功运行这个工具的例子（现在我们知道 master 和 slave 配置都正确，并且它们之间的复制也没问题）：

```
$ mysqlrplcheck --master=root:root@localhost:13001
--slave=root:root@localhost:13002
```

```
# localhost 上的 master: .....已连接。
```

```
# localhost 上的 slave: .....已连接。
```

Test Description	Status
Checking for binary logging on master	[pass]
Are there binlog exceptions?	[pass]
Replication user exists?	[pass]
Checking server_id values	[pass]
Checking server_uuid values	[pass]
Is slave connected to master?	[pass]
Check master information file	[pass]
Checking InnoDB compatibility	[pass]
Checking storage engines compatibility	[pass]
Checking lower_case_table_names settings	[pass]
Checking slave delay (seconds behind master)	[pass]

```
# .....完成。
```

我们已经有了复制配置工具和复制检查工具，如果还能以图形的方式看到复制拓扑就太棒了。下面这个工具就可以显示拓扑情况。其核心功能对高可用实用工具非常重要，接下来这一小节将讨论这个工具。



## 显示拓扑结构：mysqlrplshow

还有一个有用的复制工具能够查看拓扑图。显示复制工具 *mysqlrplshow* 显示给定 master 上连接的 slave。slave 必须以 `--report-host` 和 `--report-port` 启动，才能在 SHOW SLAVE HOSTS 视图中显示。

如果提供了 `--recurse` 选项，实用工具会试图连接每个 slave 并显示这个 slave 的 slave。通过这种方式，实用工具可以发现所有可达服务器的拓扑，其中认证信息由 `--discover-slaves-login` 选项提供。



对那些使用默认认证信息连接失败的 slave，使用 `--prompt` 选项为每个 slave 提示输入用户名和密码。

这个工具还能识别环形复制拓扑。如果使用 `--show-list` 选项，该工具能够以不同的格式显示服务器的列表。这些格式包括 GRID、CSV、TAB 和 VERTICAL。输出结果还可以导入到另一个应用程序脚本，以便进一步分析或报告。

### mysqlrplshow 示例

下面的例子展示了在一个简单的层次型复制拓扑中运行实用工具。注意，slavehost2 上有一个中间 master。

```
$ mysqlrplshow --master=root:pass@masterhost:13001 --recurse \
  --discover-slaves-login=root:root --format=grid --show-list
# masterhost 上的 master: .....已连接。
# 为 master 查找 slave: localhost:13001
# slavehost1 上的 master: .....已连接。
# 为 master 查找 slave: localhost:13002
# slavehost2 上的 master: .....已连接。
# 为 master 查找 slave: localhost:13003
# slavehost3 上的 master: .....已连接。
# 为 master 查找 slave: localhost:13004

# 复制拓扑图
masterhost:13001 (MASTER)
|
+--- slavehost1:13002 - (SLAVE)
|
+--- slavehost2:13003 - (SLAVE + MASTER)
|
+--- slavehost3:13004 - (SLAVE)
```

Master	Slave
masterhost:13001	slavehost1:13002
masterhost:13001	slavehost2:13003
slavelhost:13002	slavehost3:13004

## 高可用的实用工具

MySQL 实用工具中有两个专门用于高可用（HA）解决方案的实用工具。一个是执行日常操作和按需操作的常规工具 *mysqlrpladmin*，另一个是不间断运行用来监控 master 及其 slave 的工具 *mysqlfailover*。

下面几个小节将具体介绍这些工具及其示例，并讨论这些工具的使用场景。

### 概念

首先我们讨论一下 HA 实用工具中用到的一些概念。与很多专用工具一样，通常在使用工具之前需要用户理解一些思想和概念。例如，HA 工具执行两个基本操作，即正常切换（switchover）和故障转移（failover）。下面我们将帮助你理解工具的模式、命令和输出，从而更好地使用这些工具。有些内容可能很熟悉了，不过这里还是回顾一下。



我们假定你已经看完了前面关于复制和高可用性的相关章节，包括第 4、5、6 章，并且已经对复制术语有了很好的理解。

### 全局事务标识符

全局事务标识符（Global Transaction Identifier, GTID）是添加到二进制日志中的特殊标记，具有唯一值，用于标识事件的开始。这些标记被添加在每组事件（即事务）之前，由服务器的唯一全局标识符（Unique Universal Identifier, UUID）和一个序列数字构成。这个序列数是顺序生成的，从 1 开始。

因此，GTID 促成了一种新的复制协议，即 slave 能够向 master 请求任何尚未执行的 GTID。确定某个 GTID 是否已经被应用的计算过程简单而准确，因为 GTID 是唯一的。

这种新概念和新协议使这件事成为可能：如果某个 master 宕机，选择它的任何一个 slave，通过收集来自其他 slave 的 GTID，可以将这个 slave 作为新的 master。因此，

MySQL 第一次拥有了一种有助于故障转移的原生机制。

如果你计划运行 *mysqlfailover* 工具并在 *mysqlrpladmin* 工具中执行故障转移命令，那么必须启用 GTID (*gtid\_mode=ON* 选项)。

## 候选 slave

候选 slave (candidate slave) 是当 master 宕机或不可达的时候，用于代替 master 的那个 slave。候选 slave 的选择可以是自动的，需要满足以下全部条件：

- slave 正在运行并可达。
- 启用了 GTID。
- slave 没有落后于 master。
- slave 的复制过滤没有冲突。具体来说，过滤器的数据库列表都能够匹配。例如，如果 master 的 *--binlog-do-db* 列举了 *db1* 和 *db3*，slave 的 *--replicate-do-db* 列举了 *db1* 和 *db2*，那么这些过滤器就会冲突，因为复制到 slave 上的数据与从 master 过滤来的数据并不一样。
- slave 上存在复制用户。
- slave 上启用了二进制日志。

## slave 选举

这是选择一个候选 slave 代替 master 的过程。实用工具有一个功能能够将这个步骤作为故障转移的前兆。

## 故障转移

故障转移是在 master 宕机或不可达的时候替换 master 角色的过程。

## 正常切换

正常切换是有目的地将 master 从一个活动的 master 切换到另一个 slave。这不同于故障转移，因为这个时候 master 是可用的，新 slave 可以从 master 获取全部事件从而保持最新状态。例如，当不得不出于维护目的使 master 离线的时候，就可以执行正常切换。

## mysqlrpladmin

*mysqlrpladmin* 工具允许管理员在 MySQL 复制资源上按需执行操作（即运行一个命令然后退出）。可用的命令包括：

elect

选举最佳 slave，得到优胜者。

652 failover

将 master 故障转移到最佳 slave。

gtid

显示全局事务标识符变量的状态，并显示所有服务器的 UUID。

health

显示 master 及其 slave 的复制健康状况。

reset

在每个 slave 上发出 STOP SLAVE 和 RESET SLAVE 命令。

start

启动所有 slave。

stop

停止所有 slave。

switchover

执行 slave 提升。

下面提供一些常用命令的例子，以及这些命令的具体使用方法。

## mysqlrpladmin 示例

首先我们来看使用 health 命令的复制管理工具。这个命令显示连接到 master 的 slave 列表，以及 slave 的状态和复制健康状况。下面的例子显示了一个 master 及其 slave。注意，我们使用了发现选项（-disco），传递用于连接各个 slave 的用户名和密码。如果想看更多健康报告的详细信息，可使用 --verbose 选项。

```
$ mysqlrpladmin --master=root:pass@localhost:13002 --disco=root:pass health
```

```
# 为 localhost:13002 上的 master 发现 slave
```

```
# 检查权限。
```

```
#
```

```
# 复制拓扑健康状况：
```

host	port	role	state	gtid_mode	health
localhost	13002	MASTER	UP	ON	OK
localhost	13003	SLAVE	UP	ON	OK



```
| localhost | 13004 | SLAVE | UP | ON | OK |
| localhost | 13005 | SLAVE | UP | ON | OK |
+-----+-----+-----+-----+-----+
# .....完成。
```



记住，必须使用 `--report-host` 和 `--report-port` 选项启动 slave，才能使用发现 slave 这个功能。

◀ 653

如果想使用 `failover` 命令而不是故障转移工具手动进行故障转移，`elect` 命令能够预见选择哪个 slave 用于故障转移。在构建容错应用的时候，如果你需要知道新 master 的具体参数，这就很有用。下面的例子展示了如何确定选择哪个 slave：

```
$ mysqlrpladmin --master=root:pass@localhost:13002 --disco=root:pass elect
# 为 localhost:13002 上的 master 发现 slave
# 检查权限。
# 从已知 slave 中选举候选 slave。
# 最佳 slave 位于 localhost:13003。
# .....完成。
```

GTID 命令显示拓扑中全部 GTID 的列表。如果 master 和 slave 已经运行了很长时间，并且有很多事务，那么这个列表将会很长。这个报告有助于诊断事务问题。下面是 GTID 命令的输出片段：

```
$ mysqlrpladmin --master=root:pass@localhost:13002 --disco=root:pass gtid
# 为 localhost:13002 上的 master 发现 slave
# 检查权限。
#
# 所有服务器的 UUID:
```

host	port	role	uuid
localhost	13002	MASTER	673d40e4-32ac-11e2-87f6-419d4fb292c5
localhost	13003	SLAVE	7cb05e3e-32ac-11e2-87f6-336560b463d4
localhost	13004	SLAVE	940abdc-32ac-11e2-87f7-9f158c80ded6
localhost	13005	SLAVE	a680695c-32ac-11e2-87f7-797d07ab4557

```
#
# 服务器上运行的事务:
```

host	port	role	gtid
localhost	13002	MASTER	0035A78A-32AF-11E2-8807-6F5662E76235:1-2

```
| localhost | 13002 | MASTER | 42077312-32AC-11E2-87F5-4FD43CAEB376:1-2 |
| localhost | 13002 | MASTER | 498880AE-32B1-11E2-8815-C19FD75A3D21:1-2 |
| localhost | 13002 | MASTER | 4AA055C2-32AF-11E2-8808-366E88604744:1-2 |
| localhost | 13002 | MASTER | 59D25A04-32AF-11E2-8809-6D4E4056F0AF:1-2 |
| localhost | 13002 | MASTER | 5D3211AC-32B0-11E2-880F-F6735C743197:1-2 |
| localhost | 13002 | MASTER | 673D40E4-32AC-11E2-87F6-419D4FB292C5:1-3 |
| localhost | 13002 | MASTER | 74151132-32AE-11E2-8803-E93E7AABEB97:1-2 |
| localhost | 13002 | MASTER | 7CB05E3E-32AC-11E2-87F6-336560B463D4:1-3 |
.....
| localhost | 13005 | SLAVE | 7CB05E3E-32AC-11E2-87F6-336560B463D4:1-3 |
| localhost | 13005 | SLAVE | 940ABCD-32AC-11E2-87F7-9F158C80DED6:1-3 |
| localhost | 13005 | SLAVE | 9E6957E4-32AF-11E2-880B-C19ED44E0636:1-2 |
| localhost | 13005 | SLAVE | A680695C-32AC-11E2-87F7-797D07AB4557:1-3 |
| localhost | 13005 | SLAVE | C903D9AC-32AF-11E2-880C-6D4F415B0402:1-2 |
| localhost | 13005 | SLAVE | F37C4048-32AF-11E2-880D-CFD6C34C3629:1-2 |
+-----+-----+-----+-----+
# .....完成。
```

复制管理工具最常用的命令是 `switchover` 命令。这个命令将 master 角色从当前 master 以可控的方式切换到某个特定 slave。我们说“可控”是因为 slave 能够在角色发生改变之前保持与 master 同步。

这个命令有一个很方便的选项, `--demote-master` 选项。这个选项告诉 `switchover` 命令: 一旦角色改变完成, 就将原来的 master 变成新 master 的一个 slave。这样, 就可以在一组 slave 之间轮换 master 角色。下面给出了运行这个命令的例子。注意, 运行的末尾给出了健康状况报告, 可用来验证角色变化。比较一下这个健康报告和之前例子中的健康报告。

花点时间仔细看一下这个输出。切换过程的所有步骤都有简单的状态语句说明。

```
$ mysqlrpladmin --master=root:pass@localhost:13003 --disco=root:pass failover
# 为 localhost:13003 上的 master 发现 slave
# 检查权限。
# 执行故障转移。
# localhost:13002 上的候选 slave 即将成为新的 master。
# 候选 slave 准备故障转移。
# 如果复制用户不存在则创建复制用户。
# 停止 slave。
# 在所有 slave 上执行 STOP。
# 将 slave 切换为新的 master。
# 启动 slave。
# 在所有 slave 上执行 START。
# 检查 slave 是否有错误。
# 故障转移完成。
```

```
#
# 复制拓扑健康状况：
+-----+-----+-----+-----+-----+-----+
| host      | port | role  | state | gtid_mode | health |
+-----+-----+-----+-----+-----+-----+
| localhost | 13002 | MASTER | UP    | ON        | OK     |
| localhost | 13004 | SLAVE  | UP    | ON        | OK     |
| localhost | 13005 | SLAVE  | UP    | ON        | OK     |
+-----+-----+-----+-----+-----+-----+
# .....完成。
```

我们最后才讲 failover 命令，是为了展示如何在不使用故障转移工具的情况下手动完成故障转移。故障转移工具能够自动完成故障转移。下面是运行 failover 命令的结果。注意，这个命令在操作的末尾部分还打印了健康报告。

花点时间仔细看一下这个输出。切换过程的所有步骤都有简单的状态语句说明。

```
$ mysqlrpladmin --master=root:pass@localhost:13003 --disco=root:pass failover
# 为 localhost:13003 上的 master 发现 slave
# 检查权限。
# 执行故障转移。
# localhost:13002 上的候选 slave 即将成为新的 master。
# 候选 slave 准备故障转移。
# 如果复制用户不存在则创建复制用户。
# 停止 slave。
# 在所有 slave 上执行 STOP。
# 将 slave 切换为新的 master。
# 启动 slave。
# 在所有 slave 上执行 START。
# 检查 slave 是否有错误。
# 故障转移完成。
#
# 复制拓扑健康状况：
+-----+-----+-----+-----+-----+-----+
| host      | port | role  | state | gtid_mode | health |
+-----+-----+-----+-----+-----+-----+
| localhost | 13002 | MASTER | UP    | ON        | OK     |
| localhost | 13004 | SLAVE  | UP    | ON        | OK     |
| localhost | 13005 | SLAVE  | UP    | ON        | OK     |
+-----+-----+-----+-----+-----+-----+
# .....完成。
```

## mysqlfailover

故障转移工具并不是那种一次性运行完毕的工具，而是交互式运行的。故障转移工具又称故障转移控制台（failover console），是一个纯文本的命令行工具。

控制台能够在任何系统上运行，但应该运行在那些可以密切监视的系统上。如果在 master 发生故障的时候，而运行自动故障转移工具的那个服务器正好离线了，那这个故障转移策略就不那么顺利了。此外，如果拓扑中有多个 master，则需要运行一个（只需要一个）工具实例，并且连接一个且只要一个 master。

控制台仅用于启用 GTID 的服务器，不适合不支持 GTID 的服务器。

656 由于这是一个交互性工具，启动控制台以后就有很多菜单命令可供选择。我们来看一下控制台连接一个附有 4 个 slave 的 master 是什么样子：

```
$ mysqlfailover --master=root:pass@localhost:13001 --disco=root:pass
--log=failover_log.txt
```

```
# 为 localhost:13001 上的 master 发现 slave
```

```
# 检查权限。
```

```
[...]
```

```
MySQL Replication Failover Utility
```

```
Failover Mode = auto      Next Interval = Mon Nov 19 20:00:49 2012
```

```
Master Information
```

```
-----
```

```
Binary Log File  Position Binlog_Do_DB Binlog_Ignore_DB
clone-bin.000001 574
```

```
GTID Executed Set
```

```
42077312-32AC-11E2-87F5-4FD43CAEB376:1-2
```

```
Replication Health Status
```

```
+-----+-----+-----+-----+-----+-----+
| host      | port  | role   | state | gtid_mode | health |
+-----+-----+-----+-----+-----+-----+
| localhost | 13001 | MASTER | UP    | ON        | OK     |
| localhost | 13002 | SLAVE  | UP    | ON        | OK     |
| localhost | 13003 | SLAVE  | UP    | ON        | OK     |
| localhost | 13004 | SLAVE  | UP    | ON        | OK     |
| localhost | 13005 | SLAVE  | UP    | ON        | OK     |
+-----+-----+-----+-----+-----+-----+
```

```
Q-quit R-refresh H-health G-GTID Lists U-UUIDs L-log entries
```





上面的 [...] 表示屏幕刷新。故障转移控制台是一个文本命令窗口(终端)。因此,“MySQL Replication Failover Utility”一行显示在窗口的第一行。

控制台的顶部显示了 master 信息,包括当前日志文件和位置,以及 GTID 执行组。然后是健康报告,与复制管理工具 *mysqlrpladmin* 中的一样。

屏幕底部是菜单。使用标签前面的大写字母选择以下各项:

#### Quit

退出控制台。

#### Refresh

657

刷新健康报告。在查看健康报告的时候,使用这个命令刷新下一个时间间隔的数据。

#### Health

显示健康报告。使用这个命令切换到健康报告视图。

#### GTID Lists

显示 master 上执行组的 GTID、slave 的已执行 GTID、slave 的已清除 GTID 和 slave 拥有的 GTID。按 G 键在所有列表间反复循环。使用这个命令将视图切换到 GTID 列表。

#### UUID

显示所有服务器的 UUID。使用这个命令将视图切换到 UUID 列表。

#### Log Entries

读取并显示故障转移日志文件中的行。只有启用了二进制日志才会显示这一项。

故障转移控制台以固定时间间隔检查 master 的状态。默认是每隔 15 秒。使用 `--interval` 选项可以更改这个频率。

故障转移控制台有一个有趣的功能,对进行的操作生成日志。日志中保存了每条语句和状态项。使用 `--log-file` 选项开启日志,指定日志文件的路径和文件名。日志文件默认以追加而不是重写的方式生成。不过可以使用 `--log-age=N` 选项告诉工具覆盖超过 *N* 天以上的旧记录。

当需要永久记录事件,或者想要更高级别的审计功能的时候,日志文件就非常有用。

还有一个有用的选项, `--rediscover`。这个选项告诉命令在每个时间间隔重新运行 slave 发现功能。如果要用故障转移工具检测新 slave 是什么时候添加的,就可以使用这个选项。

## 故障转移模式

故障转移控制台有几种操作模式，能够以最适合你的环境或需求的方式运行故障转移控制台。下面描述了每种模式（默认模式是 AUTO）。

### AUTO

自动将故障转移到候选者。如果没有可用的候选 slave，则从 slave 列表中继续定位可用的候选者。如果找不到可用的候选者，报错并退出。

658

一旦找到候选者，将故障转移到最佳 slave。这个命令会测试每一个符合要求的候选 slave。一旦选举了某个候选 slave，将它作为其他各个 slave 的 slave，从而收集除候选者以外其他 slave 上运行的事务。这样，候选者就成了最新的 slave。

### ELECT

这个模式与 AUTO 一样，但是如果候选列表中没有可用的候选 slave，它不再检查剩下的 slave，而是报错并退出。如果想要使用某个 slave 子集完成故障转移，并且剩下的 slave 都不符合要求，那么可以使用这个模式。例如，你可能要排除那些硬件比较旧或较差的 slave。

### FAIL

在 master 失效的时候报错但不执行故障转移。这个模式用于提供周期性健康监控，并不执行故障转移操作。

## 故障转移事件

当 master 既不能通过数据库连接可达也无法 ping 通的时候，就发生了故障转移事件。每隔一段时间就要检查一次。如果检查失败，故障转移事件即被触发。

故障转移检测也是可调的。使用 `--ping` 选项告诉工具在认为 master 离线之前要 ping 多少次。使用外部脚本可以控制故障转移的检测。本章后面“扩展点”一节解释了如何使用外部脚本触发故障转移事件。

下面总结了故障转移事件的执行步骤。这是默认故障转移模式的过程。根据模式选择不同，slave 选举稍有不同（参见本章前面“故障转移模式”一节）：

- 新的候选 slave 按下面的方式选举。如果指定了 `--candidate-slaves` 选项，就按照 slave 的出现顺序测试它们。如果没有符合要求的 slave（参见本章前面“候选 slave”一节），并且指定了 `--slaves` 选项，那么按照 slave 的出现顺序测试它们。如果这些 slave 都没有被选举，工具就会测试每一个它能够发现的 slave。由于这些条件要求并不严格，很可能 `--candidate-slaves` 选项列出的第一个 slave 就是第一个候选 slave。

- 一旦选举了候选 slave，它就成为其他每个 slave 的 slave。使用新的复制协议，保证候选 slave 能够获取除了该候选 slave 自身以外其他 slave 上执行的任何事务。
- 现在候选 slave 上拥有全部事务并保持同步了，成为新的 master。
- 最后，剩下的 slave 连接到新 master，并启动复制。

执行上述过程的时候，故障转移控制台暂时切换到列表模式，即列出每个步骤的状态语句。如果使用了 `--verbose` 选项，则工具会显示操作的所有细节。

下面的例子是故障转移事件的执行过程。注意，你可以看到全部 GTID 甚至 `CHANGE MASTER` 命令。如果启用了日志功能，这些语句全部都还会出现在日志文件中：

```
Failover starting in 'auto' mode...
# 检查 localhost:13002 上的 slave 是否可以作为候选 slave。
# GTID_MODE=ON...OK
# 复制用户存在 ...ok
# localhost:13002 上的候选 slave 将会成为新的 master。
# 准备故障转移。
# 加锁: FLUSH TABLES WITH READ LOCK
# 将候选 slave 作为 master 连接到 localhost:13003，获取未处理的 GTID。
# 为 localhost:13002 执行更改 master 的命令
# CHANGE MASTER TO MASTER_HOST = 'localhost', MASTER_USER = 'rpl',
MASTER_PASSWORD = 'rpl', MASTER_PORT = 13003, MASTER_AUTO_POSITION=1
# 取消锁: UNLOCK TABLES
# 等待候选 slave 与 localhost:13003 上的 slave 保持一致。
# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('0035A78A-32AF-11E2-8807-
6F5662E76235:1-2', 3)
# 返回 code = 0
# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('42077312-32AC-11E2-87F5-
4FD43CAEB376:1-2', 3)
# 返回 code = 0
# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('498880AE-32B1-11E2-8815-
C19FD75A3D21:1-2', 3)
# 返回 code = 0
# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('74151132-32AE-11E2-8803-
E93E7AABEB97:1-2', 3)
# 返回 code = 0
# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('7CB05E3E-32AC-11E2-87F6-
336560B463D4:1-3', 3)
# 返回 code = 0
```

```

# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('9E6957E4-32AF-11E2-880B-
C19ED44E0636:1-2', 3)
# 返回 Code = 0
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('F37C4048-32AF-11E2-880D-
CFD6C34C3629:1-2', 3)
# 返回 Code = 0
# 加锁: FLUSH TABLES WITH READ LOCK
# 将候选 slave 作为 master 连接到 localhost:13004, 获取未处理的 GTID。
# 为 localhost:13002 执行更改 master 命令
# CHANGE MASTER TO MASTER_HOST = 'localhost', MASTER_USER = 'rpl',
MASTER_PASSWORD = 'rpl', MASTER_PORT = 13004, MASTER_AUTO_POSITION=1
# 取消锁: UNLOCK TABLES
# 等待候选 slave 与 localhost:13004 上的 slave 保持一致。
# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('0035A78A-32AF-11E2-8807-
6F5662E76235:1-2', 3)
# 返回 Code = 0
# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('42077312-32AC-11E2-87F5-
4FD43CAEB376:1-2', 3)
# 返回 Code = 0
# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('498880AE-32B1-11E2-8815-
C19FD75A3D21:1-2', 3)
# 返回 Code = 0
# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('74151132-32AE-11E2-8803-
E93E7AABEB97:1-2', 3)
# 返回 Code = 0
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('940ABCDC-32AC-11E2-87F7-
9F158C80DED6:1-3', 3)
# 返回 Code = 5
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('9E6957E4-32AF-11E2-880B-
C19ED44E0636:1-2', 3)
# 返回 Code = 0
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('F37C4048-32AF-11E2-880D-
CFD6C34C3629:1-2', 3)
# 返回 Code = 0
# 加锁: FLUSH TABLES WITH READ LOCK
# 将候选 slave 作为 master 连接到 localhost:13005, 获取未处理的 GTID。

```



```

# 为 localhost:13002 执行更改 master 命令
# CHANGE MASTER TO MASTER_HOST = 'localhost', MASTER_USER = 'rpl',
MASTER_PASSWORD = 'rpl', MASTER_PORT = 13005, MASTER_AUTO_POSITION=1
# 取消锁: UNLOCK TABLES
# 等待候选 slave 与 localhost:13005 上的 slave 保持一致。
# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('0035A78A-32AF-11E2-8807-
6F5662E76235:1-2', 3)
# 返回 Code = 0
# localhost:13002 上的 slave:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('42077312-32AC-11E2-87F5-
4FD43CAEB376:1-2', 3)
# 返回 Code = 0
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('498880AE-32B1-11E2-8815-
C19FD75A3D21:1-2', 3)
# 返回 Code = 0
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('4AA055C2-32AF-11E2-8808-
366E88604744:1-2', 3)
# 返回 Code = 5
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('59D25A04-32AF-11E2-8809-
6D4E4056F0AF:1-2', 3)
# 返回 Code = 0
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('5D3211AC-32B0-11E2-880F-
F6735C743197:1-2', 3)
# 返回 Code = 0
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('74151132-32AE-11E2-8803-
E93E7AABEB97:1-2', 3)
# 返回 Code = 0
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('9E6957E4-32AF-11E2-880B-
C19ED44E0636:1-2', 3)
# 返回 Code = 0
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('A680695C-32AC-11E2-87F7-
797D07AB4557:1-3', 3)
# 返回 Code = 0
# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('C903D9AC-32AF-11E2-880C-
6D4F415B0402:1-2', 3)
# 返回 Code = 0

```

```

# Slave localhost:13002:
# QUERY = SELECT SQL_THREAD_WAIT_AFTER_GTIDS('F37C4048-32AF-11E2-880D-
CFD6C34C3629:1-2', 3)
# 返回 Code = 0
# 如果复制用户不存在,则创建复制用户。
# 停止 slave。
# 在所有 slave 上执行 STOP。
# localhost:13002 上的 slave 执行 STOP ok
# localhost:13003 上的 slave 执行 STOP ok
# localhost:13004 上的 slave 执行 STOP ok
# localhost:13005 上的 slave 执行 STOP ok
# 将 slave 切换到新 master。
# 为 localhost:13002 执行更改 master 命令
# CHANGE MASTER TO MASTER_HOST = 'localhost', MASTER_USER = 'rpl',
MASTER_PASSWORD = 'rpl', MASTER_PORT = 13002, MASTER_AUTO_POSITION=1
# 为 localhost:13003 执行更改 master 命令
# CHANGE MASTER TO MASTER_HOST = 'localhost', MASTER_USER = 'rpl',
MASTER_PASSWORD = 'rpl', MASTER_PORT = 13002, MASTER_AUTO_POSITION=1
# 为 localhost:13004 执行更改 master 命令
# CHANGE MASTER TO MASTER_HOST = 'localhost', MASTER_USER = 'rpl',
MASTER_PASSWORD = 'rpl', MASTER_PORT = 13002, MASTER_AUTO_POSITION=1
# 为 localhost:13005 执行更改 master 命令
# CHANGE MASTER TO MASTER_HOST = 'localhost', MASTER_USER = 'rpl',
MASTER_PASSWORD = 'rpl', MASTER_PORT = 13002, MASTER_AUTO_POSITION=1
# 启动 slave。
# 在所有 slave 上执行 START。
# localhost:13003 上的 slave 执行 START ok
# localhost:13004 上的 slave 执行 START ok
# localhost:13005 上的 slave 执行 START ok
# 检查 slave 的错误。
# localhost:13003 状态: ok
# localhost:13004 状态: ok
# localhost:13005 状态: ok
# 故障转移完成。
# 为 localhost:13002 上的 master 发现 slave。

```

故障转移控制台将在 5 秒内重启。

一旦完成故障转移,控制台就会重新获取含有新 master 信息及其新 slave 列表的健康报告。根据前面的例子,故障转移之后的健康报告如下所示:

MySQL Replication Failover Utility

Failover Mode = auto      Next Interval = Mon Nov 19 20:28:03 2012

## Master Information

-----  
Binary Log File Position Binlog\_Do\_DB Binlog\_Ignore\_DB  
clone-bin.000001 7073

## GTID Executed Set

0035A78A-32AF-11E2-8807-6F5662E76235:1-2 [...]

## Replication Health Status

host	port	role	state	gtid_mode	health
localhost	13002	MASTER	UP	ON	OK
localhost	13003	SLAVE	UP	ON	OK
localhost	13004	SLAVE	UP	ON	OK
localhost	13005	SLAVE	UP	ON	OK

Q-quit R-refresh H-health G-GTID Lists U-UUIDs L-log entries

注意，原来位于端口 13001 的 master 不再显示，选举的 slave 成为新的 master，所有 slave 连接到新 master。

## 扩展点 (Extension point)

663

为了使故障转移工具更加灵活，它还支持 4 个扩展点，用来调用你自己的脚本。这样，就可以将容错应用调整为通过故障转移控制台进行交互。下面列出了可用的扩展点及其用法（按照它们在故障转移事件中的出现顺序列出）：

### -exec-fail-check

该选项允许你提供一个应用程序特定的故障转移事件，即用自定义的事件检测代替默认的故障转移检测标准（master 无法可达且无法 ping 通）。返回代码 0 表示故障转移不应该发生。非 0 的返回码表示应该进行故障转移。每次时间间隔开始时运行这个脚本。在这种情况下不使用超时选项。

### --exec-before

在故障转移事件发生之前，实用工具运行这个脚本。该选项告诉应用程序停止所有的写操作，并准备好更改 master。

### --exec-after

在故障转移事件发生之后、slave 切换到新 master 之前，实用工具运行这个脚本。该选项告诉应用程序新 master 已经准备好了。

--exec-post-failover

在 slave 重定向到新 master 并且生成了健康报告之后，实用工具运行这个脚本。该选项告诉应用程序可以重新读取 slave 了。

## 创建自己的实用工具

现在我们知道了什么是 MySQL 实用工具，以及如何用于复制，你可能对这些工具的新用法甚至特定环境下使用新工具有一些想法。幸运的是，MySQL 实用工具已经设计了这种扩展性。这一节将讨论 MySQL 实用工具的架构和组织化，还将讨论创建自定义工具或使用模板自定义复制工具的示例。

## MySQL 实用工具的结构

实用工具是通过单个脚本构建的，该脚本负责定义和验证选项和参数。脚本（即实用工具名）引用一系列模块，包含 Python 类和执行操作的方法，称为命令模块（command modules）。它们是构建更多服务器的特定组件或行为的 Python 类和方法的接口。我们将那些包含这些类和方法的文件称为公用模块（common modules）。命令模块和公用模块都使用 Connector/Python 连接器来连接 MySQL 服务器。

因此，MySQL 实用工具库是所有命令模块和公用模块的集合。这些文件都存储在同一个文件夹下。如果下载了 MySQL 实用工具的源代码，将会看到这些文件夹：`/scripts`、`/mysql/utilities/command`、`/mysql/utilities/common`。

图 17-3 显示了 MySQL 实用工具各部分之间的关系，以及每个工具是如何构建的。例如，`mysqldbcopy` 脚本在 `/scripts/mysqldbcopy.py` 文件中实现，调用名为 `/mysql/utilities/command/dbcopy.py` 的命令模块，该模块调用位于 `/mysql/utilities/common` 文件夹下的 `server.py`、`options.py`、`database.py`。

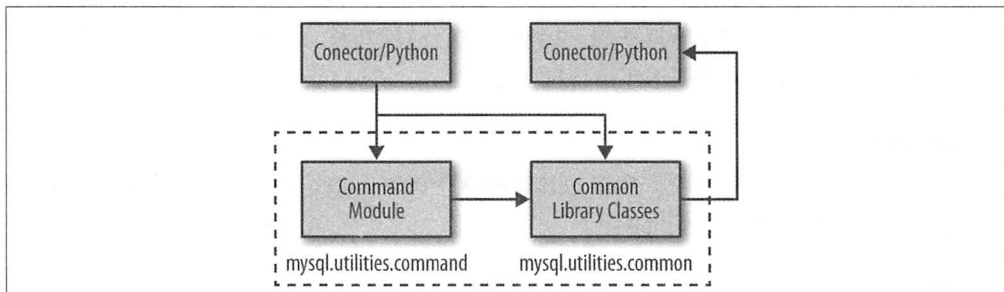


图17-3 MySQL实用工具的结构



## 自定义工具的示例

这一节介绍一个通过集成已有工具的功能来自定义工具的示例。假定我们需要一个从某个服务器向另一个服务器复制全部数据库和用户的工具。而且，我们希望这个工具能够创建新的服务器实例，并将它作为复制的目标。这个工具还非常有助于创建开发和数据问题诊断的测试环境。

首先我们简单列出需要的操作，并仔细看一下每个操作的代码。根据源代码我们总结出了一个完整的操作列表。

### 操作

如果不考虑管理性操作，比如定义选项和验证参数及其值，我们可以创建新工具的简单操作列表。主要包括三部分。第一，需要连接服务器，并确定需要复制哪些内容。第二，需要克隆哪个服务器。最后，需要在新服务器上执行复制操作。我们将这个操作列表扩展成 8 个步骤：

665

1. 配置参数。
2. 连接到原始服务器。
3. 寻找所有数据库。
4. 寻找所有用户。
5. 克隆原始服务器。
6. 连接到新服务器。
7. 复制所有数据库。
8. 复制所有用户。

如果考虑已有的可用工具，可以使用复制数据库工具、复制（克隆）用户工具，以及一个能够克隆正在运行或离线服务器的服务器。根据实用工具的结构，我们很容易看到这个解决方案由三个工具组成。实际上，我们可以研究这些工具，找到它们所用的命令模块，然后在新工具中重建同样的调用就可以了。

下一节详细解释每一个操作。

### 样本代码

首先配置实用工具的参数。因为我们重用了其他工具的部分功能，所以要研究这些工具是如何创建的，并找出那些我们需要用到的选项。

如果不考虑那些附加功能，我们只需要几个选项。最小化这个列表，我们发现并不需要那么多选项。服务器克隆工具需要 `--new-data`、`--new-port` 和 `--new-id`。数据库复制工具需要使用 `--databases` 选项列出数据库。所有工具都要有 `--server` 选项，才能连接原始服务器进行复制和克隆操作。还要使用 `--users` 选项找到需要复制的用户。

现在我们有了操作列表，打开一个名为 `cloneserver.py` 的新文件，从其他工具中复制选项。需要用到的选项如下所示：

```
parser = optparse.OptionParser(
    version=VERSION_FRM.format(program=os.path.basename(sys.argv[0])),
    description=DESCRIPTION,
    usage=USAGE,
    add_help_option=False)
parser.add_option("--help", action="help")

# 配置实用工具的选项

# 源服务器的连接信息
parser.add_option("--server", action="store", dest="server",
    type="string", default="root@localhost:3306",
    help="connection information for original server in " + \
    "the form: user:password@host:port:socket")

# 新实例的数据目录
parser.add_option("--new-data", action="store", dest="new_data",
    type="string", help="the full path to the location "
    "of the data directory for the new instance")

# 新实例的端口
parser.add_option("--new-port", action="store", dest="new_port",
    type="string", default="3307", help="the new port "
    "for the new instance - default=%default")

# 新实例的服务器 ID
parser.add_option("--new-id", action="store", dest="new_id",
    type="string", default="2", help="the server_id for "
    "the new instance - default=%default")

# 数据库列表
parser.add_option("-d", "--databases", action="store", dest="dbs_to_copy",
    type="string", help="comma-separated list of databases "
    "to include in the copy (omit for all databases)",
    default=None)
```

```
# 用户列表
parser.add_option("-u", "--users", action="store", dest="users_to_copy",
                  type="string", help="comma-separated list of users "
                  "to include in the copy (omit for all users)",
                  default=None)
```



MySQL 实用工具的当前版本使用 optparse 模块代替 argparse。

接下来的操作是连接服务器，其实现过程如下。注意，我们必须导入一个帮助方法来解析服务器选项的值，然后导入 Server 类，构建选项的字典，最后调用 connect() 方法。使用 Python 字典存储选项，是所有实用工具的习惯做法：

```
# 解析连接信息
from mysql.utilities.common.options import parse_connection
try:
    conn = parse_connection(opt.server)
except:
    parser.error("Server connection values invalid or"
                "cannot be parsed.")
```

```
# 连接原始服务器
from mysql.utilities.common.server import Server

server_options = {'conn_info' : conn, 'role' : "source"}
srv1 = Server(server_options)
srv1.connect()
```

现在我们连接到了 MySQL 服务器，就可以构建数据库列表了。如果用户想要复制所有数据库，我们要检查数据库选项，如果为空，则获取服务器上的全部数据库列表。用户的处理也是一样。下面是如何实现这一点的例子：

```
# 如果没有在选项中指定，获取服务器的数据库列表
print "# Getting databases..." db_list = []
if opt.dbs_to_copy is None:
    for db in server1.get_all_databases():
        db_list.append((db[0], None))
else:
    for db in opt.dbs_to_copy.split(","):
        db_list.append((db, None))
```

```

# 获取服务器的所有用户
print "# Getting users..."
user_list=[]
if opt.users_to_copy is None:
    users = server1.exec_query("SELECT user, host "
                                "FROM mysql.user "
                                "WHERE user != 'root' and user != ''")

    for user in users:
        user_list.append(user[0]+'@'+user[1])
else:
    for user in opt.users_to_copy.split(","):
        user_list.append(user)

```

下一步是克隆服务器。因为我们仅考虑了最少的选项集合，这里考虑几种默认值。如果需要的话，多做几个选项也不难。下面的例子是克隆代码。因为这是第一个调用命令模块的操作，所以会有一些复杂，但也很简单。实用工具的命令模块具有很高（有时又称宏观级）的封装度，所以要得到功能强大的命令模块和脚本文件很容易。

正如你可能会怀疑的一样，这个代码跟原始服务器的连接代码很像。下面就是这个操作的代码：

```

# 解析源连接的值
try:
    conn = parse_connection(opt.server)
except:
    parser.error("Server connection values invalid or cannot be parsed.")

# 获得服务类实例
print "# Connecting to server..."
server_options = {
    'conn_info' : conn,
    'role'      : "source",
}
server1 = Server(server_options)
try:
    server1.connect()
except UtilError, e:
    print "ERROR:", e.errmsg

```

也许你想知道怎么处理错误。实用工具大量使用了异常处理。对于那些已知错误的常规操作，命令模块返回一个典型的返回码，即 False 或 True。但是如果发生灾难性错误，比如无效的用户认证信息，库模块就抛出一个异常。

现在我们准备复制前面识别到的数据库和用户。每个操作的命令模块都需要服务器连接：



```

print "# Copying databases..."
try:
    dbcopy.copy_db(conn, dest_values, db_list, options)
except exception.UtilError, e:
    print "ERROR:", e.errmsg
    exit(1)

# 构建选项字典
options = {
    "overwrite" : True,
    "quiet"      : True,
    "globals"    : True
}

print "# Cloning the users..."
for user in user_list:
    try:
        res = userclone.clone_user(conn, dest_values, user,
                                   (user,), options)
    except exception.UtilError, e:
        print "ERROR:", e.errmsg
        exit(1)

```

这就是新工具的全部代码了！让我们来看看它的完整代码，及其执行示例。

## 放到一起

◀ 669

既然我们已经看到每个操作的代码了，下面来看一下完整的源代码。示例 17-3 给出了全部样本代码，以及一些典型的管理和配置代码。

示例17-3：完整的源代码

```

import optparse
import os
import sys

from mysql.utilities import VERSION_FRM
from mysql.utilities.command import dbcopy
from mysql.utilities.command import serverclone
from mysql.utilities.command import userclone
from mysql.utilities.common.server import Server
from mysql.utilities.common.options import parse_connection
from mysql.utilities import exception

# 常量
NAME = "example - copy_server "

```

```

DESCRIPTION = "copy_server - copy an existing server"
USAGE = "%prog --server=user:pass@host:port:socket " \
        "--new-dir=path --new-id=server_id " \
        "--new-port=port --databases=db list " \
        "--users=user list"

# 配置命令解析器
parser = optparse.OptionParser(
    version=VERSION_FRM.format(program=os.path.basename(sys.argv[0])),
    description=DESCRIPTION,
    usage=USAGE,
    add_help_option=False)
parser.add_option("--help", action="help")

# 配置工具的选项

# 源服务器的连接信息
parser.add_option("--server", action="store", dest="server",
                  type="string", default="root@localhost:3306",
                  help="connection information for original server in " + \
                  "the form: user:password@host:port:socket")

# 新实例的数据目录
parser.add_option("--new-data", action="store", dest="new_data",
                  type="string", help="the full path to the location "
                  "of the data directory for the new instance")

# 新实例的端口
parser.add_option("--new-port", action="store", dest="new_port",
                  type="string", default="3307", help="the new port "
                  "for the new instance - default=%default")

# 新实例的服务器 ID
parser.add_option("--new-id", action="store", dest="new_id",
                  type="string", default="2", help="the server_id for"
                  "the new instance - default=%default")

# 数据库列表
parser.add_option("-d", "--databases", action="store", dest="dbs_to_copy",
                  type="string", help="comma-separated list of databases"
                  "to include in the copy (omit for all databases)",
                  default=None)

# 用户列表
parser.add_option("-u", "--users", action="store", dest="users_to_copy",

```

```

        type="string", help="comma-separated list of users"
        "to include in the copy (omit for all users)",
        default=None)

# 现在我们处理剩下的参数
opt, args = parser.parse_args()

# 解析源连接的值
try:
    conn = parse_connection(opt.server)
except:
    parser.error("Server connection values invalid or cannot be parsed.")

# 获取服务器类的实例
print "# Connecting to server..."
server_options = {
    'conn_info' : conn, 'role': "source",
}
server1 = Server(server_options)
try:
    server1.connect()
except UtilError, e:
    print "ERROR:", e.errmsg

# 如果选项中未指定, 获取服务器的数据库列表
print "# Getting databases..."
db_list = []
if opt.dbs_to_copy is None:
    for db in server1.get_all_databases():
        db_list.append((db[0], None))
else:
    for db in opt.dbs_to_copy.split(","):
        db_list.append((db, None))

# 获取服务器的所有用户列表
print "# Getting users..."
user_list=[]

if opt.users_to_copy is None:
    users = server1.exec_query("SELECT user, host "
                               "FROM mysql.user "
                               "WHERE user != 'root' and user != ''")
for user in users:
    user_list.append(user[0]+'@'+user[1])

```

```

else:
    for user in opt.users_to_copy.split(","):
        user_list.append(user)

# 克隆服务器
print "# Cloning server instance..."
try:
    res = serverclone.clone_server(conn, opt.new_data, opt.new_port,
                                    opt.new_id, "root", None, False, True)
except exception.UtilError, e:
    print "ERROR:", e.errmsg
    exit(1)

# 设置连接的值
dest_values = {
    "user" : conn.get("user"),
    "passwd" : "root",
    "host" : conn.get("host"),
    "port" : opt.new_port,
    "unix_socket" : os.path.join(opt.new_data, "mysql.sock")
}

# 构建选项的字典
options = {
    "quiet" : True,
    "force" : True
}

print "# Copying databases..."
try:
    dbcopy.copy_db(conn, dest_values, db_list, options)
except exception.UtilError, e:
    print "ERROR:", e.errmsg
    exit(1)

# 构建选项的字典
options = {
    "overwrite" : True,
    "quiet" : True,
    "globals" : True
}

print "# Cloning the users..."
for user in user_list:
    try:

```



```

        res = userclone.clone_user(conn, dest_values, user,
                                   (user,), options)
    except exception.UtilError, e:
        print "ERROR:", e.errmsg
        exit(1)

print "# ...done."

```

示例 17-4 是这个例子的输出。我们看到新工具用于复制两个数据库和所有用户，并创建了新服务器。

示例17-4：执行示例

```

$ copy_server --new-port=3307 --server=root:pass@localhost \
  --new-data=/Volumes/Source/testabc -d db1,db2
# 连接服务器.....
# 获取数据库.....
# 获取用户.....
# 克隆服务器实例.....
# 克隆 localhost 上正在运行的 MySQL 服务器
# 创建新的数据目录.....
# 配置新实例.....
# 定位 mysql 工具.....
# 配置空数据库和 mysql 表.....
# 启动服务器的新实例.....
# 测试连接新实例.....
# 成功!
# 设置 root 密码.....
# 连接信息:
# .....完成
# 复制数据库.....
# 复制用户.....
# .....完成

```

## 建议

这个例子非常有助于对已有服务器创建一个正在运行的克隆。但是，如果要在复制的时候完成，如果能够将新服务器作为原始服务器的克隆，那么这个工具将更加有用。实际上，这样的话新工具就可以用于提供 slave 了。这多酷啊！

为此，需要在复制工具中使用命令模块。下面的例子是你需要用到的代码线索。具体改写留给读者自行完成。首先确定需要哪些选项，然后将它们添加到文件中：

```

from mysql.utilities.command.setup_rpl import setup_replication

```

...

```
673 res = setup_replication(m_values, s_values, opt.rpl_user,  
options, opt.test_db)
```

## 小结

MySQL 复制是一个强大的功能，当服务器数量较少的时候，其配置和管理很简单。如果拓扑中有很多服务器，或者复杂性较高，比如分层型拓扑或地理位置上分散的远程站点，或者通过过滤进行数据隔离配置，那么复制的管理将是一个挑战。

这一章我们介绍了管理复制服务器操作的一些实用工具，包括复制数据库、克隆用户、配置和检查复制。还介绍了使用单个命令实现正常切换和提供自动故障转移的实用工具。

我们已经了解了如何使用 MySQL 实现高可用性，现在可以使用 MySQL 实用工具进一步增强你的 MySQL 配置。

Joel 按下快捷键将最新版的管理 MySQL 复制的声明保存了下来。“搞定。”在完成工作的时候，他满意地说。

Joel 的新邮件提醒响了。他有些惊恐地看了一下收件箱。跟他怀疑的一样，老板在询问他写得怎么样了。他点击回复，草拟了一个简单的回复，然后把新文件作为附件加上去。

还没来得及点击发送，门口一阵敲门声分散了他的注意力。“Joel，4号线上有个人懂 MySQL。就用你写的那个新文档，考考他复制方面的知识，怎么样？”老板说完就匆匆消失了，Joel 发送了那封邮件，然后接起了电话。“他是怎么做到的？”Joel 疑惑道。

# 复制的提示和技巧

本附录总结了运行、诊断、修复和改善 MySQL 复制的有用提示和技巧。这是补充资料，因此可能并不包含全部细节信息，不足以作为一个完整的教程。更多细节请查询在线 MySQL 复制手册。

这几小节描述 MySQL 提供的一些功能，但在编写本书的时候这些功能还没有被官方发布。

## 使用 verbose 选项检查二进制日志

如果使用的是基于行的日志，可以使用 `--verbose` 选项查看事件的查询重组。下面给出了基于行的日志的二进制日志的检查情况：

```
$ mysqlbinlog --verbose master-bin.000001
BINLOG '
qZnvSRMBAAAAKQAAAAYCAAAAABAAAAAAAAABHRLc3QAAnQxAAEDAAE=
qZnvSRcBAAAAJwAAAC0CAAAQABAAAAAAAEAAf/+AwAAP4EAAAA  '/*!*/;
### INSERT INTO test.t1
### SET
### @1=3
### INSERT INTO test.t1
### SET
### @1=4
```

注意，那些通过 `@n` 格式给出的字段值。基于行的复制按照字段位置传递并应用记录，但忽略字段的名字。

## 使用复制重新填充表

676

如果 slave 上的表损坏了，可能是发生错误或者意外（例如用户删除了数据），那么可

以使用复制恢复数据。在 master 上创建一个临时表，作为原始表的副本，并将原始表删除，然后从副本中重建数据。只要这个操作不影响到其中任何字段的数据类型（比如 autoincrement），那么就没问题。这就是利用复制重新填充 slave 上的表。这个过程有两种形式，取决于使用哪一种日志。

## 基于语句的日志

如果使用的是基于语句的日志，在每个表上运行以下代码：

```
SELECT * INTO OUTFILE 't1.txt' FROM t1;  
DROP TABLE IF EXISTS t1;  
CREATE TABLE t1 ...;  
LOAD DATA INFILE 't1.txt' INTO TABLE t1;
```

## 基于行的日志

如果使用的是基于行的日志，临时表并不会被转移到 slave 中去。因为，只要发送引导表所需的数据即可，使用 INSERT INTO 命令：

```
CREATE TEMPORARY TABLE t1_tmp LIKE t1;  
INSERT INTO t1_tmp SELECT * FROM t1;  
DROP TABLE IF EXISTS t1;  
CREATE TABLE t1 SELECT * FROM t1_tmp;
```

## 使用 MySQL 代理进行多主复制

slave 的 master 不能多于一个，但是一个 master 却可以有很多 slave。大部分时候这不是问题，但是如果需要合并两个不同 master 的数据，然后将这个合成数据复制到 slave 中的话，怎么办呢？

方法之一就是使用 MySQL 代理作为中间 slave。图 A-1 展示了这个配置的概念图。

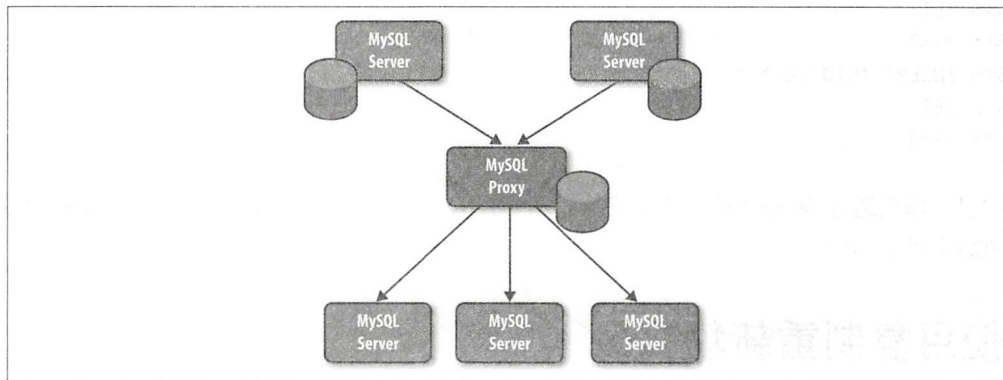


图 A-1: 使用MySQL代理处理多个master



MySQL 代理同时从两个 master 接收变更（事件），并在不保存数据（写入数据库）的情况下写入新的二进制日志。slave 将 MySQL 代理作为 master 使用。这样就可以合并两个 master 的数据，然后复制到一组 slave 中去，这就有效地实现了多主复制。

关于 MySQL 代理的更多信息，请参见在线 MySQL 手册的 MySQL 代理一节。

## 使用默认存储引擎

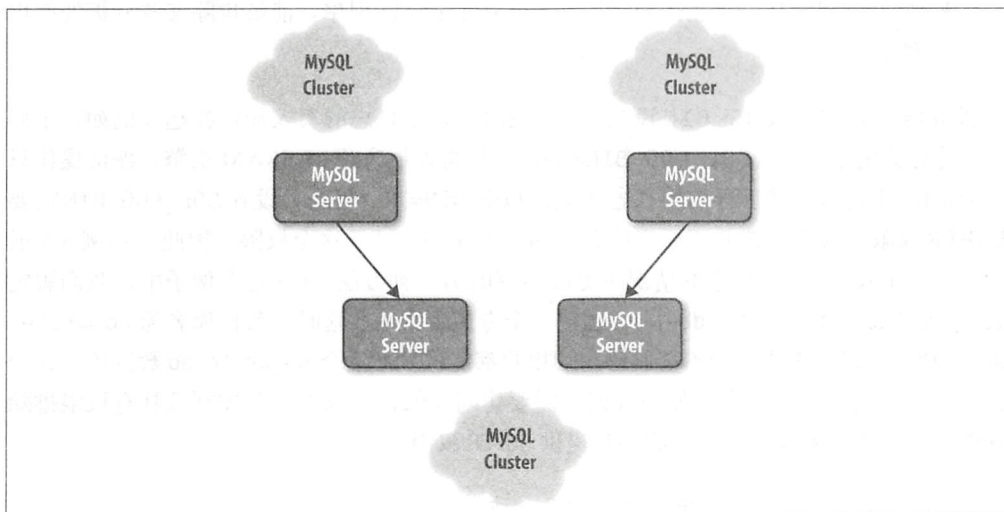
如果在 master 上使用默认存储引擎，即发出 SET GLOBAL STORAGE ENGINE 命令，那么这个命令不会被复制。因此，任何没有指定存储引擎的 CREATE 语句都将获得 slave 上设置的默认存储引擎。所以，InnoDB 表可能会被复制到 MyISAM 表中。如果忽略这些警告，可能会遇到不兼容性导致的复制停止的情况。

一个解决方法是保证每个 slave 上都设置了全局存储引擎，即在配置文件中设置 default-storage-engine 选项。但是，最佳的办法是在 CREATE 语句上指定存储引擎。

这样并不能避免一个问题，即 master 上使用的存储引擎在 slave 上不存在。这时，只能在 slave 上安装缺少的存储引擎来解决问题。

## MySQL 集群多源复制

使用 MySQL 集群可以配置多源复制（参见图 A-2）。为此，将 master 配置为复制不同的数据。通过将数据放置在不同的 master 上，就可以避免键的交叉影响和类似的上下文相关的值。

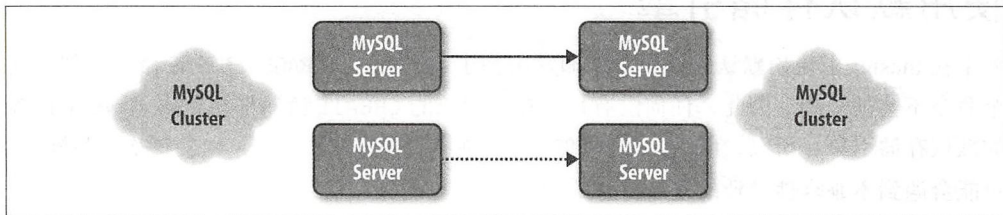


图A-2：多源复制

## 带有故障转移的多通道复制

为了获得更好的可靠性和快速恢复，可以使用双复制配置（参见图 A-3），即包好两组复制拓扑，可以从一个集群复制到另一个集群。如果一边失效了，可以迅速将故障转移到另一边。

注意，服务器的名字格式为 *mysql.X*，保存的位置为 *mysql.X.savepos*。



图A-3: 多通道复制

679

## 使用当前数据库过滤

二进制日志过滤器能够非常方便地按照特定目的摒弃语句。

例如，有些语句只适合服务器，比如设置表的引擎，使 master 和 slave 上使用不同的引擎。slave 上要用不同引擎的原因可能包括：

- MyISAM 更适合做报表，而且二进制日志包含完整的事务，所以可以在 master 上使用 InnoDB 进行事务处理，然后在 slave 上使用 MyISAM 进行报表和分析处理。
- 为了节省内存，通过在 slave 上使用 Blackhole 引擎，能够排除某些分析处理用的表。

将服务器变量 `SQL_LOG_BIN` 设为 0，可以将二进制日志的写入操作挂起。例如，示例 A-1 首先关闭了日志（`SQL_LOG_BIN=0`），然后将表更改为 MyISAM 引擎，保证操作只在 master 上进行，然后再启用日志（`SQL_LOG_BIN=1`）。但是，设置 `SQL_LOG_BIN` 需要 SUPER 权限，我们可能并不愿意给普通的数据库用户授予这个权限。因此，示例 6-8 展示了在 master 上关闭日志的情况下更改引擎的另一种方法。（在这个例子中，我们假定表 `my_table` 在数据库 `my_db` 中）。创建一个专用数据库（这时，数据库名为 `no_write_db`），然后，任何想要执行不复制语句的用户都能够 USE 这个 `no_write_db` 数据库，这个语句将会被过滤出去。由于使用数据库需要访问权限，所以你可以控制谁具有权限排除这些语句，而不需要向不信任的用户提供 SUPER 权限。

示例A-1: 过滤二进制日志中的语句的两种方法

```
master> SET SQL_LOG_BIN = 0;
```

Query OK, 0 rows affected (0.00 sec)

master> ALTER TABLE my\_table ENGINE=MyISAM;

Query OK, 92 row affected (0.90 sec)

Records: 92 Duplicates: 0 Warnings: 0

680

master> SET SQL\_LOG\_BIN = 1;

Query OK, 0 rows affected (0.00 sec)

master> USE no\_write\_db;

Reading table information for completion of table and column names

You can turn off this feature to get a quicker startup with -A

Database changed

master> ALTER TABLE my\_db.my\_table ENGINE=MyISAM;

Query OK, 92 row affected (0.90 sec)

Records: 92 Duplicates: 0 Warnings: 0

## slave 上的字段比 master 上的多

如果 slave 上有一些 master 上没有的额外字段——例如记录的时间戳，或者添加路由或其他位置数据——可以向 slave 上的表添加字段，而不需要向 master 添加字段。MySQL 复制能够忽略这些额外字段。在 slave 上真正将数据插入这些额外字段的时候，将这些字段定义为接受默认值（比如，插入时间戳的简单方法），或者使用一个定义在 slave 上的触发器提供值。

对于基于语句的日志来说，可按照以下方式创建字段。

1. 在 master 上创建表：

```
CREATE TABLE t1 (a INT, b INT);
```

2. 在 slave 上更改表：

```
ALTER TABLE t1 ADD ts TIMESTAMP;
```

3. 在 master 上插入：

```
INSERT INTO t1(a,b) VALUES (10,20);
```

对于基于行的日志，必须在行的末尾添加这些新字段，并且要有默认值。如果在行的中间或者前面添加字段的话，基于行的复制就会失败。只要在表的末尾添加字段并且提供了默认值，就不需要考虑复制哪些语句，因为基于行的复制会从正在进行更新、插入或删除操作的行中直接提取字段。

# slave 上的字段比 master 上的少

681

如果 slave 上的字段比 master 上的少——例如，要保护敏感数据，或者减少复制的数据量——可以从 slave 的表中删除字段，而不需要删除 master 上的字段。MySQL 基于行的复制会忽略这些丢失的字段。但是，slave 上删掉的字段必须来自 master 上行的末尾。

对于基于行的复制来说，按照以下方式删除字段。

- 1. 在 master 上创建表：

```
CREATE TABLE t1 (a INT, b INT, comments TEXT);
```

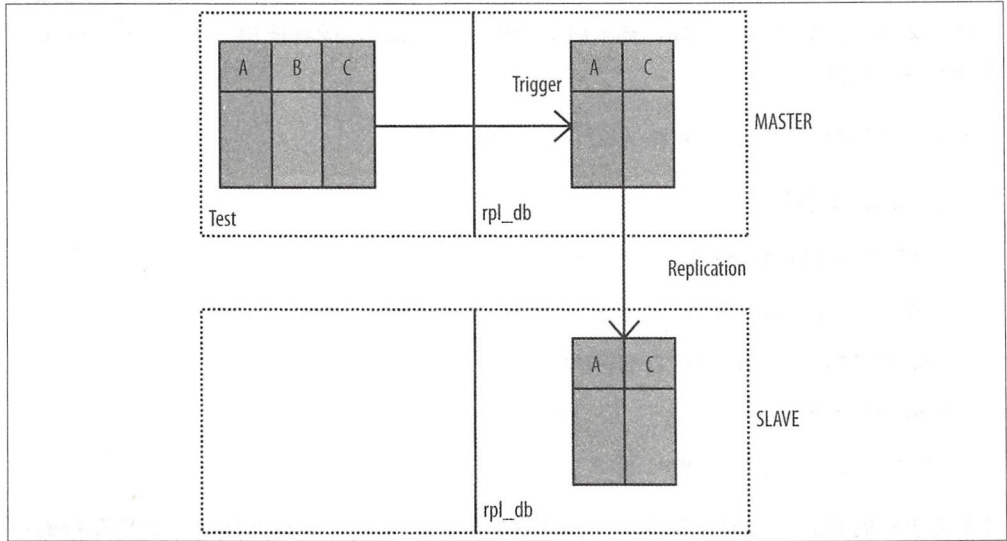
- 2. 在 slave 上更改表：

```
ALTER TABLE t1 DROP comments;
```

- 3. 在 master 上插入：

```
INSERT INTO t1 VALUES (1,2,"Do not store this on slave");
```

如果在 master 上使用触发器更新某个隔离数据库中的基表，就只有 slave 数据库中的字段被复制。这样，基于行的复制就能够复制任意子集的字段，不管这些字段是否位于末尾。图 A-4 展示了这个方法的概概念图。这里，表中的三个字段通过触发器转换到复制数据库中的另一个表。



图A-4：复制字段的子集

下面的例子是如何使用触发器更新复制数据库中的表。在 master 上，执行以下操作：



```

CREATE DATABASE rpl_db;
USE test;
CREATE TABLE t1 (a INT, b BLOB, c INT);
CREATE TRIGGER tr_t1 AFTER INSERT ON test.t1 FOR EACH ROW
INSERT INTO rpl_db.t1_v(a,c) VALUES(NEW.a,NEW.c);
USE rpl_db;
CREATE TABLE t1_v (a INT, c INT);

```

在执行正常插入的时候，触发器就会提取字段子集，然后写入 rpl\_db 数据库中的表。然后这个数据库被复制，但原始的表不会被复制：

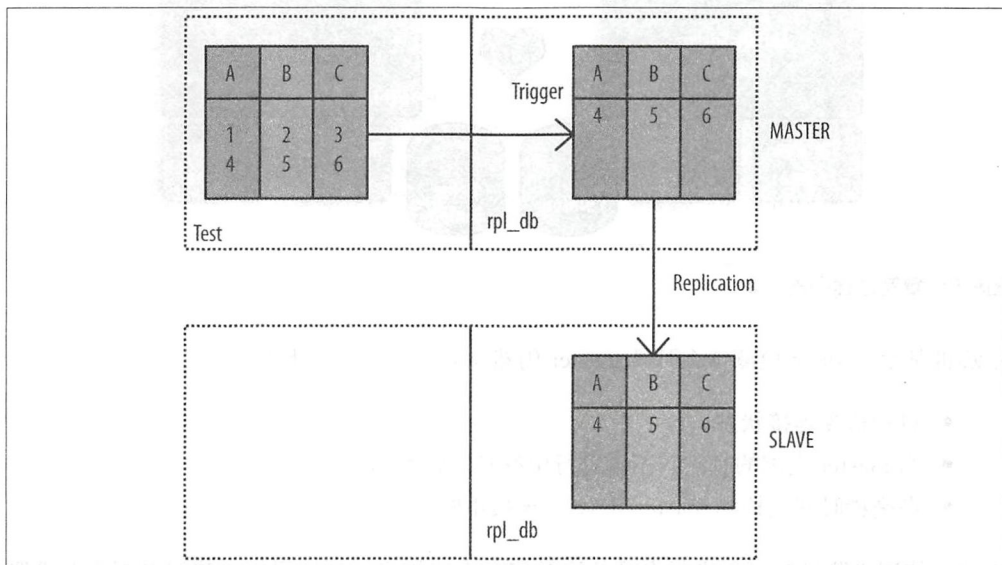
```

USE test;
SET @blob = REPEAT( 'beef' ,100);
INSERT INTO t1 VALUES (1,@blob,3), (2,@blob,9);

```

## 将选定行复制到 slave

还可以进行分块复制，只有满足一定条件的行才会被复制。该技术从 master 向 slave 复制一个特定的数据库（通过配置过滤器实现），并且由触发器来决定哪些行需要被复制。触发器决定了哪些行需要被复制，然后向复制表中插入这些行（参见图 A-5）。



图A-5: 复制行的子集

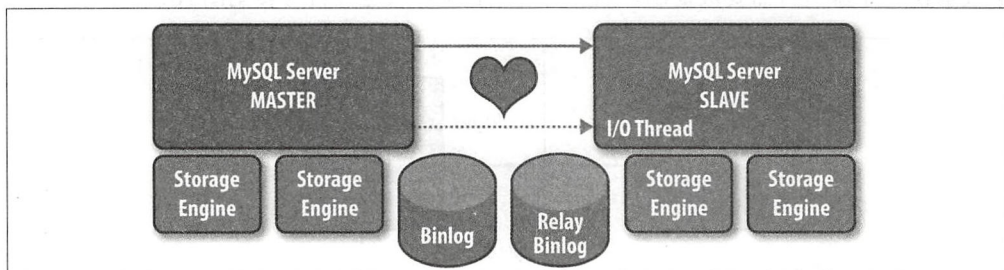
下面这个例子展示了如何使用触发器更新复制数据库中的表，只有字段 a 的值为奇数的那些行才被插入：

683 # 仅复制那些第一个字段的值为奇数的行

```
USE rpl_db;
CREATE TABLE t1_h (a INT, b BLOB, c INT);
--delimiter //
CREATE TRIGGER slice_t1_horiz AFTER INSERT ON test.t1
FOR EACH ROW
BEGIN
    IF NEW.a MOD 2 = 1 THEN INSERT
        INTO rpl_db.t1_h VALUES (NEW.a, NEW.b, NEW.c);
    END IF;
END//
--delimiter ;
```

## 复制心跳（5.4.4 以及更新的版本）

使用复制心跳机制，可以提升复制的可靠性和故障转移。如果启用了心跳，slave 就会周期性地询问 master。只要 slave 得到答复，就照常继续在下一个时间间隔询问 master。服务器维护接收到的心跳数目的统计数据，有助于确定 master 是否离线或者无响应。图 A-6 展示了心跳机制的概念图。



图A-6: 复制过程中的心跳

心跳机制允许 slave 检测什么时候 master 仍然响应，具有以下优点：

- 自动检查连接状态
- 当 master 空闲的时候，不再进行中继日志轮换
- 在毫秒时间内检测 master/slave 的连接中断

因此，使用心跳机制可以进行自动故障转移和类似的高可用性操作。使用以下命令设置 slave 上的心跳间隔：

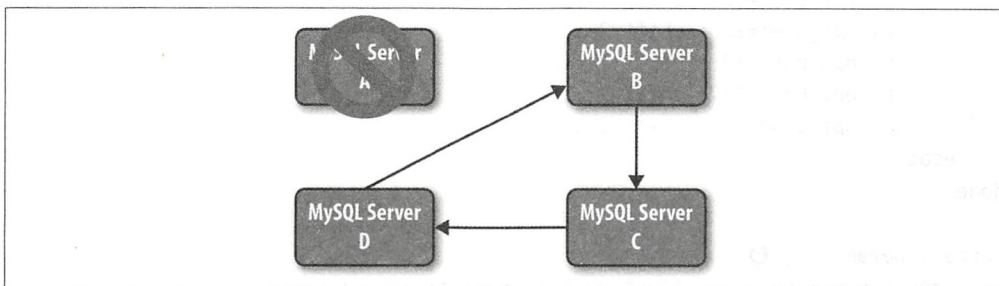
684 CHANGE MASTER SET master\_heartbeat\_period= val;

以下命令能够检查心跳机制的配置和统计数据：

```
SHOW STATUS like 'slave_heartbeat period';  
SHOW STATUS like 'slave_received_heartbeats';
```

## 在环形复制中忽略服务器（5.5 以及更新的版本）

如果是环形复制拓扑，并且其中一个服务器失效了，那么必须改变拓扑以排除这个失效的服务器。在图 A-7 中，服务器 A 失效了，必须将它从环形拓扑中删除。这个时候，可以通过设置服务器 B 来终结服务器 A 上的事件，形成新环。



图A-7：在环形复制中替换服务器

为此，在服务器 B 上发出以下命令：

```
CHANGE MASTER TO MASTER_HOST=C ... IGNORE_SERVER_IDS=(A)
```

这个功能在 MySQL 5.5 中被引入。

## 延时复制（5.6 及更新的版本）

有时候需要延迟复制——例如，当你要确保 slave 并没有由于过多变更而负荷过载。这个功能允许以这种方式复制：总是比 master 迟  $n$  秒才在 slave 上复制事件。可像下面这样设置参数：

```
CHANGE MASTER TO MASTER_DELAY= seconds
```

其中 *seconds* 是一个非负整数，这是一个小于 MAX\_ULONGLONG 的数，表示 slave 等待的秒数（即由于错误被拒绝进行尝试的最大次数）。

运行 SHOW SLAVE STATUS 命令，检查 Seconds\_behind\_master 字段的输出，可以验证延迟的配置。

关于延时复制的更多内容请阅读 MySQL 参考手册（<http://bit.ly/delay-rep>）。

## 常见任务的 shell 命令

这一节的 shell 函数对于你的 shell 脚本有很大的帮助。这个库可以包含在脚本文件中，解析那些来自命令行的命令行选项。

```
set -- `getopt 'S:u:P:h:p:' "$@"` ❶
while [ x$1 != x-- ]
do
    case $1 in
        -S) opt_sock=$2 ; shift 2 ;;
        -u) opt_user=$2 ; shift 2 ;;
        -P) opt_port=$2 ; shift 2 ;;
        -h) opt_host=$2 ; shift 2 ;;
        -p) opt_pass=$2 ; shift 2 ;;
    esac
done

connect_param () { ❷
    echo --user=$opt_user ${opt_pass:++-password="$opt_pass"} \
        ${opt_sock:++-socket="$opt_sock"} \
        ${opt_host:++-host="$opt_host"}    ${opt_port:++-port="$opt_port"}
}

mysql_exec () { ❸
    echo "$@" |
    mysql `connect_param` --vertical --batch
}

stop_slave () {
    mysql_exec STOP SLAVE $2
}

start_slave () {
    mysql_exec START SLAVE $2
}

change_master () {
    host=${1:+MASTER_HOST=\ '$1\ '}
    port=${2:+MASTER_PORT=$2}
    user=${3:+MASTER_USER=\ '$3\ '}
    pass=${4:+MASTER_PASSWORD=\ '$4\ '}
    file=${5:+MASTER_LOG_FILE=\ '$5\ '}
    pos=${6:+MASTER_LOG_POS=$6}
    mysql_exec CHANGE MASTER TO \
        $host ${host:+,} $port ${port:+,} \
```



```

    $user ${user:+,} $pass ${pass:+,} \
    $file ${file:+,} $pos
}

fetch_slave_exec_pos () {
    mysql_exec SHOW SLAVE STATUS | ❷
    grep '\<Relay_Master_Log_File\|\<Exec_Master_Log_Pos' |
    cut -f2 -d:
}

fetch_slave_read_pos () {
    mysql_exec SHOW SLAVE STATUS |
    grep '\<Master_Log_File\|\<Read_Master_Log_Pos' |
    cut -f2 -d:
}

fetch_master_pos () {
    mysql_exec SHOW MASTER STATUS |
    grep '\<File\|\<Pos' | cut -f2 -d:
}

slave_wait_for_empty_relay_log () {
    stop_slave IO_THREAD
    fetch_slave_read_pos | {
        read file pos
        mysql_exec "SELECT MASTER_POS_WAIT('$file', $pos)"
    }
    stop_slave SQL_THREAD
}

```

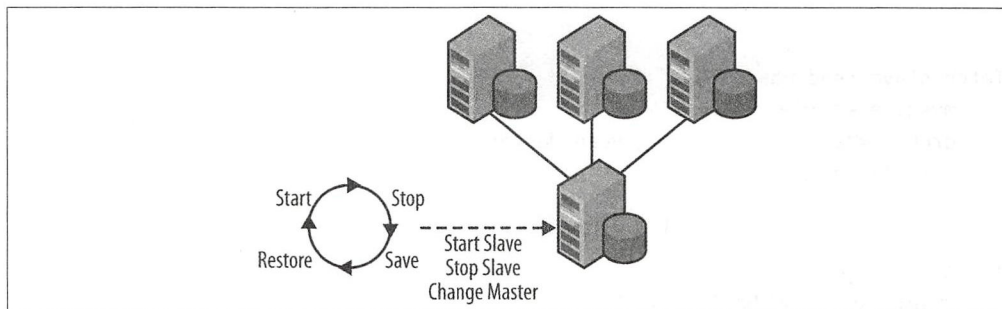
- ❶ 使用标准 UNIX 命令 *getopt* 读取选项。使用 *source* 引入这个库，然后从命令行读取选项。它通过 *-h* 指定主机名，*-p* 指定端口，*-u* 指定用户，*-p* 指定密码。
- ❷ 使用 *mysql* 命令需要使用打印连接参数的函数。所有同 *mysql* 命令的参数一样的命令都可以使用这个函数，所以可通过 *mysqladmin connection\_param shut down* 关闭服务器。
- ❸ 帮助函数简化了 SQL 命令的执行。使用前面设置的连接信息，假定命令使用全部参数。

由于这个函数能够用于其他 shell 函数，所以跳过字段名，并且向标准输出输出简单的多栏式结果。要进一步处理这个结果，必须使用如 *cut* 将列分割开来。

687 ④ 这里我们看到了如何处理结果集的例子。在这个例子中，含有最后一个执行事务的文件和位置行被排除，然后使用 *cut* 抽取字段和值。

## 用 shell 脚本进行多源复制

MySQL slave 不能配置成从多个 master 读取变更，但是可以通过一种时间共享的方式模拟，即客户端以轮盘的方式向 slave 发出 SQL 命令，如图 A-8 所示。



图A-8：使用轮盘方式的多master

这个过程的实现有几种不同的方式，根据你使用什么语言以及服务器版本而不同。一般来说，过程如下：

1. 初始化，为所有 master 存储日志位置和日志文件。这个位置应该是你想要启动复制的位置。
2. 选择其中一个 master 作为当前 master（current master）。
3. 重复以下步骤：
  - a. 使用 `CHANGE MASTER` 将 slave 设置为从当前 master 读取。
  - b. 使用 `START SLAVE` 启动 slave。
  - c. 让复制运行一段时间，从当前 master 执行复制。
  - d. 使用 `STOP SLAVE` 停止复制。
  - e. 使用 `SHOW SLAVE STATUS` 将复制的当前位置信息保存起来。

下面使用本附录前面“常见任务的 shell 命令”一节中的 shell 函数库实现客户端，使用 shell 脚本以及一些额外的函数，展示这个简单实现。当然，这个过程也可以使用其他语

言实现。这段代码将位置数据存储在文件中，其中每个文件基于 slave 服务器的名字按照数字顺序命令：

```

fetch_host_and_pos () { ❶
    mysql_exec $1 SHOW SLAVE STATUS |
    grep '\<Master_\(Host\|Port\|Log_File\)\|\<Read_Master_Log_Pos' |
    cut -f2 -d:
}

stop_and_save () { ❷
    sock="/var/run/mysqld/$1.sock"
    stop_slave $socket
    fetch_host_and_pos $sock >$1.savepos
}

restore_and_start () { ❸
    socket="/var/run/mysqld/$1.sock"
    cat $1.savepos | {
        read host
        read port
        read file
        read pos
        change_master $socket $host $port $file $pos
        start_slave $socket
    }
}

cnt=1
while true ❹
do

    stop_and_save mysqld.$cnt
    cnt=`expr $cnt % 5 + 1`
    restore_and_start mysqld.$cnt
    sleep 60
done

```

- ❶ 这是一个帮助函数，获取主机和位置，以保存到文件。因为数据是以字段的形式获取的，所以可以获得包含位置信息的行，然后抽取字段名。
- ❷ 这个函数使用 STOP SLAVE 停止 slave，通过 fetch\_host\_and\_pos 函数获取当前 master 的位置，然后将其保存到文件中。这个文件应该用于当前 master。
- ❸ 从给定文件恢复位置，使用 CHANGE MASTER 将 slave 设置到正确的位置，然后使用 START SLAVE 启动 slave。

④ 这是图 A-8 所示的循环，反复执行，并且在 master 之间切换。

## 689 使用存储过程更换 master

在为不同任务编写存储过程的时候，使用声明变量或用户变量来代替常量是非常有用的。当然，并不是所有命令都这样。例如，CHANGE MASTER 就不接受任何参数，所以为了对这个命令使用参数，就需要一点创造力。

构建一个包含语句的字符串，就可以通过字符串来准备语句，因此能够按照你喜欢的方式执行任何语句。这个技巧可用于构建任何你喜欢的语句：

```
delimiter $$ ①
CREATE PROCEDURE change_master(
    host VARCHAR(50) NOT NULL, port INT NOT NULL,
    user VARCHAR(50) NOT NULL, passwd VARCHAR(50) NOT NULL,
    file VARCHAR(50), pos LONG) ②
BEGIN
    SET @cmd = CONCAT('CHANGE MASTER TO ',
        CONCAT('MASTER_HOST = "', host, '" , '),
        CONCAT('MASTER_PORT = ', port, ' , '),
        CONCAT('MASTER_USER = "', user, '" , '),
        CONCAT('MASTER_PASSWORD = "', passwd, '"'));

    IF name IS NOT NULL AND pos IS NOT NULL THEN
        SET @cmd = CONCAT(@cmd,
            CONCAT(' , MASTER_LOG_FILE = "', name, '"'),
            CONCAT(' , MASTER_LOG_POS = ', pos));
    END IF;
    PREPARE change_master FROM @cmd; ③
    EXECUTE change_master;
    DEALLOCATE PREPARE change_master;
END $$
delimiter ; ④
```

① ④ 为了定义存储例程体，必须将分号分隔符改成其他形式。这里，使用 \$\$ 作为分隔符。因为通常存储过程体中不会出现这个符号，所以很方便，当然你可以使用任何分隔符。只要确保这个分隔符不会出现在存储例程体内部就好了。

② 文件给定的位置以及文件内部的位置可以为 NULL。这个时候，CHANGE MASTER 命令就得到任何文件或位置，所以复制就会从头开始。

③ 语句只接受一个用户变量，所以需要将字符串存储到一个用户变量中去。声明局部变量会很方便，但是，唉，语法不支持啊。



在本附录前面“用 shell 脚本进行多源复制”一节中我们看到，目前 slave 不支持同时从多个 master 复制（多源复制），但是可以通过时间共享模式从多个 master 进行复制，即按照固定的时间间隔切换当前 master。

本附录前面“用 shell 脚本进行多源复制”一节中的解决方案使用了一个独立的客户端（由 Bourne shell 实现）作为调度器，来定期更换 master。如果让服务器自己更换 master 不是更简单吗？好消息是，这是有可能的，如果你使用的是 MySQL 5.6 的话。使用 MySQL 5.6 的表处理事务型复制（参考第 8 章“事务型复制”一节），通过创建一个执行 master 更换的事件（CREATE EVENT），就可以完全用 SQL 实现时间共享的多源复制。

为此，需要遍历 master 列表。这个列表要包含所有必需的连接信息，以及切换到下一个 master 之前复制所在的位置。为了方便跟踪这些 master，还需要一个包含服务器序号（不是服务器 ID，只是算法中用到的序号）的字段。此外，如果能够在不需要停止时间调度器的情况下添加或删除这些 master，那就方便了。

表A-1: My\_Masters表的定义

字段	类型	NULL	键	默认值
server_id	int(10) unsigned	NO	PRI	NULL
host	varchar(50)	YES		NULL
port	int(10) unsigned	YES		3306
user	varchar(50)	YES		NULL
passwd	varchar(50)	YES		NULL

表 A-1 描述的表很好地符合要求。为了跟踪当前 master，还要引入 Current\_Master 表，记住当前活动的 master 序号。为了支持 master 的添加和删除，这两张表都应该是事务型的，因为同时更新两张表中的信息应该属于单个事务。此时，如果使用会话变量跟踪当前服务器的序号，就会很方便，但是每次事件开始的时候，都会产生一个新会话，因此会话变量的值并不会在两次事件之间保存。所以，我们必须将这个值保存到表中。

下面是切换 master 事件的完整定义（每一步都有解释）：

```
delimiter $$
CREATE EVENT multi_source
    ON SCHEDULE EVERY 10 SECOND DO
BEGIN
    DECLARE l_host VARCHAR(50);
    DECLARE l_port INT UNSIGNED;
    DECLARE l_user TEXT;
    DECLARE l_passwd TEXT;
    DECLARE l_file VARCHAR(50);
```

```

DECLARE l_pos BIGINT;
DECLARE l_next_idx INT DEFAULT 1; ❶

SET SQL_LOG_BIN = 0; ❷

STOP SLAVE IO_THREAD; ❸
SELECT master_log_name, master_log_pos
    INTO l_file, l_pos
    FROM mysql.slave_master_info;
SELECT MASTER_POS_WAIT(l_file, l_pos);
STOP SLAVE;

START TRANSACTION; ❹
UPDATE my_masters AS m, ❺
    mysql.slave_relay_log_info AS rli
    SET m.log_pos = rli.master_log_pos,
        m.log_file = rli.master_log_name
    WHERE idx = (SELECT idx FROM current_master);

SELECT idx INTO l_next_idx FROM my_masters ❻
    WHERE idx > (SELECT idx FROM current_master)
    ORDER BY idx LIMIT 1;

SELECT idx INTO l_next_idx FROM my_masters ❼
    WHERE idx >= l_next_idx
    ORDER BY idx LIMIT 1;

UPDATE current_master SET idx = l_next_idx;
COMMIT;

SELECT host, port, user, passwd, log_pos, log_file
    INTO l_host, l_port, l_user, l_passwd, l_pos, l_file
    FROM my_masters
    WHERE idx = l_next_idx;

CALL change_master(l_host, l_port, l_user, l_passwd, l_file, l_pos); ❽
START SLAVE;
END $$
delimiter ;

```

❶ 注意，这个变量的声明包含默认值 1，这在❽里面会用到。

❷ 禁用二进制日志，因为我们不想将任何语句写入二进制日志。因为这是一个事件，所以在执行结束的时候变量会自动重置，而且不影响其他。

- ③ 事件的下一步是在切换 master 之前停止 slave 的 I/O 线程，并清空中继日志。从 Slave\_Master\_Info 表中读取最后位置，并使用 MASTER\_POS\_WAIT 等待直到达到中继日志的末尾。
- ④ 因为我们想要变更同时自动更新的 My\_Masters 和 Current\_Master 两个表，所以将更新表的代码封装到事务中。
- ⑤ 旧 master 的位置保存到 My\_Masters。
- ⑥ 依次确定下一个 master，需要两步。SELECT 根据序号选择下一个 master。回忆一下，在调度器正在运行的时候，可以添加和删除 master，所以这些序号序列之间会有空隙。
- ⑦ 通过①中的默认值 1 进行环形处理。但是，如果已经有环了，那么序号为①的 master (l\_next\_idx 的默认值) 可能不存在。这个时候，扫描并寻找第一个大于等于 l\_next\_idx 的序号。
- ⑧ 获取到新 master 的信息之后，使用存储过程发出 CHANGE MASTER 命令，正如本附录前面“使用存储过程更换 master”一节中介绍的那样。

# 一个GTID的实现

MySQL 5.6 中引入了全局事务标识符，提供了一种唯一识别事务的方法，不管它在什么服务器上运行。5.6 的服务器处理故障转移非常简单，第 8 章“使用 GTID 进行故障转移”一节已经介绍过了。但是，如果你没有 5.6 版本的服务器，却仍然需要执行故障转移，就可以采用本附录中介绍的方法，总体上说就是，实现自定义的全局事务标识符，然后将它们写入二进制日志。其目的是实现一些 slave 提升所需的必备功能。

## 给服务器添加 GTID

需要为每个事务加标签，才能将 slave 的最后一个事务与二进制日志中的相应事件匹配起来。这个标签的内容和结构并不重要，只需要能够唯一标识事务即可，使得 master 上的每个事务都在已提升 slave 的二进制日志上。这些标签就是全局事务标识符，本附录就是要实现它们。

最简单的办法是在每个事务的末尾插入一个语句，更新一个特殊的表，以记下每个 slave 的位置。在提交每个事务之前，这个语句为向这个表中插入一个对那个事务来说唯一的数字。

标签的处理主要有两种方式：

- 扩展应用程序代码，执行语句。
- 调用存储过程，在存储过程中执行每一次提交和写标签操作。

694

因为第一种方法实现更简单，所以我们就介绍这种方法。如果对第二种方法感兴趣，请参考本附录后面的“使用存储过程提交事务”中介绍的内容。

为了实现全局事务标识符（GTID），像示例 B-1 那样创建两个表，一个是名为 Global\_



Trans\_ID 的表，用于生成序列号；另一个是名为 Last\_Exec\_Trans 的表，用于记录 GTID。

Last\_Exec\_Trans 的定义中添加了服务器标识符，以区分不同服务器上的事务提交。例如，如果在所有 slave 成功连接之前，已提升的 slave 失效了，那么区分原始 master 和已提升 slave 上的事务标识符就很重要。否则，当那些没有连接上已提升 slave 的 slave 被重定向到第二个提升 slave 的时候，可能会从错误的位置开始执行。这个例子中使用 MyISAM 定义计数表，当然也可以使用 InnoDB。

这一节中，为了保持代码简单和熟悉，我们采用纯 SQL 实现操作。后面会看到如何将这些操作作为应用程序的一部分自动实现。

示例B-1：用于生成和跟踪全局事务标识符的表

```
CREATE TABLE Global_Trans_ID (
    number INT UNSIGNED AUTO_INCREMENT PRIMARY KEY
) ENGINE = MyISAM;
```

```
CREATE TABLE Last_Exec_Trans (
    server_id INT UNSIGNED,
    trans_id INT UNSIGNED
) ENGINE = InnoDB;
```

-- 插入值为 NULL 的单个行，以便更新。

```
INSERT INTO Last_Exec_Trans() VALUES ();
```

下一步是创建向二进制日志添加全局事务标识符的过程，这样提升 slave 的程序就可以从日志中读取标识符。过程如下。

1. 向事务计数表插入一项。在这之前要关闭二进制日志，因为插入操作不应该被复制到 slave：

```
master> SET SQL_LOG_BIN = 0;
Query OK, 0 rows affected (0.00 sec)
```

```
master> INSERT INTO Global_Trans_ID() VALUES ();
Query OK, 1 row affected (0.00 sec)
```

2. 使用 LAST\_INSERT\_ID 函数获取全局事务标识符。为了简化逻辑，同时从服务器变量 server\_id 获取服务器标识符：

```
master> SELECT @@server_id as server_id, LAST_INSERT_ID() as trans_id;
```

```

+-----+-----+
| server_id | trans_id |
+-----+-----+
|          0 |      235 |
+-----+-----+
1 row in set (0.00 sec)

```

3. 将全局事务标识符插入跟踪表 `Last_Exec_Trans` 之前，将事务计数表中的记录删除以释放空间。只有 MyISAM 表需要这一步，因为它会内部跟踪最新的 `autoincrement` 值，并保存已删除行的 `autoincrement` 值。如果用的是 InnoDB，则必须小心表中的最后一个全局事务标识符。InnoDB 根据表中当前最大的 `autoincrement` 字段值确定下一个数字。

```

master> DELETE FROM Global_Trans_ID WHERE number < 235;
Query OK, 1 row affected (0.00 sec)

```

4. 打开二进制日志：

```

master> SET SQL_LOG_BIN = 1;
Query OK, 0 rows affected (0.00 sec)

```

5. 用第二步中的服务器标识符和事务标识符更新跟踪表 `Last_Exec_Trans`。这是 COMMIT 提交事务之前的最后一步：

```

master> UPDATE Last_Exec_Trans SET server_id = 0, trans_id = 235;
Query OK, 1 row affected (0.00 sec)

```

```

master> COMMIT;
Query OK, 0 rows affected (0.00 sec)

```

每个全局事务标识符都表示复制重新开始的地点。因此，每个事务都必须执行这个过程。如果某些事务没有执行，就没有正确地打标签，也就不能从那个位置开始复制了。

为了定义这些表，并配置服务器使用这些 GTID，可以使用 `Promotable` 类（参见示例 B-2）处理服务器。我们看到，这个例子重用了第 2 章“服务器角色”一节中的 `_enable_binlog` 帮助方法，并添加了 `log-slave-updates` 选项。因为可提升的 slave 需要一些特殊的表才能成为 master，所以添加这种可提升的 slave 也要添加这些表。为此，我们编写了 `_add_global_id_tables` 函数。这个函数假定这些表已经存在并且定义正确，所以不需要重新创建表。但是，`Last_Exec_Trans` 表开始时需要有一行才能保证更新正确执行，所以如果没有表已存在的警告，我们就可以创建这个表，并向其中插入一个 NULL 行。

示例B-2: 可提升slave角色的定义

```
_GLOBAL_TRANS_ID_DEF = """
CREATE TABLE IF NOT EXISTS Global_Trans_ID (
    number INT UNSIGNED NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (number)
) ENGINE=MyISAM
"""

_LAST_EXEC_TRANS_DEF = """
CREATE TABLE IF NOT EXISTS Last_Exec_Trans (
    server_id INT UNSIGNED DEFAULT NULL,
    trans_id INT UNSIGNED DEFAULT NULL
) ENGINE=InnoDB
"""

class Promotable(Role):
    def __init__(self, repl_user, master):
        self.__master = master
        self.__user = repl_user

    def _add_global_id_tables(self, master):
        master.sql(_GLOBAL_TRANS_ID_DEF)
        master.sql(_LAST_EXEC_TRANS_DEF)
        if not master.sql("SELECT @@warning_count"):
            master.sql("INSERT INTO Last_Exec_Trans() VALUES ()")

    def _relay_events(self, server, config):
        config.set('mysqld', 'log-slave-updates')

    def imbue(self, server):
        # 获取并更新配置
        config = server.get_config()
        self._set_server_id(server, config)
        self._enable_binlog(server, config)
        self._relay_event(server, config)

        # 加入新配置
        server.stop()
        server.put_config(config)
        server.start()

        # 向master添加表
        self._add_global_id_tables(self.__master)

        server.repl_user = self.__master.repl_user
```

## 使用 GTID 进行事务处理

在提交每个事务的时候也需要更新 Last\_Exec\_Trans 表。示例 B-3 用 PHP 实现了提交事务。代码采用 PHP 编写,因为这是应用代码的一部分,而且代码中不包含管理布局的内容。

示例B-3: 启动、提交和中止事务的代码

```
function start_trans($link) {
    mysql_query("START TRANSACTION", $link);
}

function rollback_trans($link) {
    mysql_query("ROLLBACK", $link);
}

function commit_trans($link) {
    mysql_select_db("common", $link);
    mysql_query("SET SQL_LOG_BIN = 0", $link);
    mysql_query("INSERT INTO Global_Trans_ID() VALUES ()", $link);
    $trans_id = mysql_insert_id($link);
    $result = mysql_query("SELECT @@server_id as server_id", $link);
    $row = mysql_fetch_row($result);
    $server_id = $row[0];

    $delete_query = "DELETE FROM Global_Trans_ID WHERE number = %d";
    mysql_query(sprintf($delete_query, $trans_id),
        $link);
    mysql_query("SET SQL_LOG_BIN = 1", $link);

    $update_query = "UPDATE Last_Exec_Trans SET server_id = %d, trans_id = %d";
    mysql_query(sprintf($update_query, $server_id, $trans_id), $link);
    mysql_query("COMMIT", $link);
}
```

然后,使用这个代码提交事务,调用函数而不是使用 COMMIT 和 ROLLBACK 命令。例如,编写一个 PHP 函数,向数据库中添加一条消息,并更新用户的消息计数:

```
function add_message($email, $message, $link) {
    start_trans($link);
    mysql_select_db("common", $link);
    $query = sprintf("SELECT user_id FROM user WHERE email = '%s'", $email);
    $result = mysql_query($query, $link);
    $row = mysql_fetch_row($result);
```



```

$user_id = $row[0];
$update_user = "UPDATE user SET messages = messages + 1 WHERE user_id = %d";
mysql_query(sprintf($update_user, $user_id), $link);

$insert_message = "INSERT INTO message VALUES (%d,'%s')";
mysql_query(sprintf($insert_message, $user_id, $message), $link);
commit_trans($link);
}

$conn = mysql_connect("/:var/run/mysqld/mysqld1.sock", "root");
add_message('mats@example.com', "MySQL Python Replicant rules!", $conn);

```

## 使用存储过程提交事务

这里给出的方法是同步服务器，以在应用程序中实现事务提交过程，即应用程序代码需要知道表名，以及如何产生和管理全局事务标识符这些复杂的事情。搞明白以后，就没有一开始那么复杂了。通常来说，在应用程序中创建函数能够相对简单地处理这些问题，应用程序可以无须了解具体细节调用这些函数。

将事务提交逻辑放入数据库服务器的另一个方法是，使用存储过程。根据具体情况不同，这种方法有时候更好些。例如，更改提交过程不需要更改应用程序代码。

为此，在存储例程中，要将Global\_Trans\_ID表中的事务标识符和服务器标识符放入用户自定义变量或者局部变量。根据选择的方法不同，二进制日志的查询会有一点差别。

使用局部变量不太可能干扰其他代码，因为用户自定义变量可以在存储例程之外查看甚至更改。这就是所谓“泄露”（leaking）。

那么，提交事务的过程如下：

```

CREATE PROCEDURE commit_trans ()
    SQL SECURITY DEFINER
BEGIN
    DECLARE trans_id, server_id INT UNSIGNED;
    SET SQL_LOG_BIN = 0;
    INSERT INTO Global_Trans_ID() values ();
    SELECT LAST_INSERT_ID() INTO trans_id,
           @@server_id INTO server_id;
    SET SQL_LOG_BIN = 1;
    INSERT INTO Last_Exec_Trans(server_id, trans_id)
        VALUES (server_id, trans_id);
    COMMIT;
END

```

所以，在应用程序代码中提交事务就很简单：

```
CALL Commit_Trans();
```

现在的任务是，将存储过程修改为扫描二进制日志查找全局事务标识符。那么，调用这个函数在二进制日志中是什么样子的呢？简单地调用 *mysqlbinlog*，我们看到：

```
# at 1724
#091129 18:35:11 server id 1 end_log_pos 1899 Query  thread_id=75
    exec_time=0    error_code=0
SET  TIMESTAMP=1259516111/*!*/;
INSERT INTO Last_Exec_Trans(server_id, trans_id)
    VALUES ( NAME_CONST('server_id',1), NAME_CONST('trans_id',13))
/*!*/;
# at 1899
#091129 18:35:11 server id 1 end_log_pos 1926 Xid = 1444
COMMIT/*!*/;
```

在输出结果中清楚地看到了服务器标识符和事务标识符。如何通过正则表达式进行匹配，就留给读者自己练习了。

## 寻找 GTID 的位置

在 slave 连接已提升 slave 并从正确位置开始复制的时候，需要确定已提升 slave 上的哪个位置是 slave 最后执行的事务。通过扫描已提升 slave 的二进制日志寻找 GTID，可以实现这一点。

使用 `SHOW MASTER LOGS` 可获取已提升 slave 的日志，然后就可以扫描日志寻找全局事务标识符。例如，在使用 *mysqlbinlog* 读取文件 *slave-3-bin.000005* 的时候，输出结果差不多如示例 B-4 所示。slave-3 在位置 596 接收到的事务（输出中的第一行加粗表示）的全局事务标识符来自于 slave-1，由 *Last\_Exec\_Trans* 表的一条 `UPDATE` 所示。

示例B-4：某个事务的mysqlbinlog命令的输出结果

```
# at 596
#091018 18:35:42 server id 1 end_log_pos 664 Query  thread_id=952  ...
SET  TIMESTAMP=1255883742/*!*/;
BEGIN
/*!*/;
# at 664
#091018 18:35:42 server id 1 end_log_pos 779 Query  thread_id=952  ...
SET  TIMESTAMP=1255883742/*!*/;
```

```

UPDATE user SET messages = messages + 1 WHERE id = 1
/*!*/;
# at 779
#091018 18:35:42 server id 1 end_log_pos 904 Query thread_id=952 ...
SET TIMESTAMP=1255883742/*!*/;
INSERT INTO message VALUES (1,'MySQL Python Replicant rules!')
/*!*/;
# at 904
#091018 18:35:42 server id 1 end_log_pos 1021 Query thread_id=952 ...
SET TIMESTAMP=1255883742/*!*/;
UPDATE Last_Exec_Trans SET server_id = 1, trans_id = 245
/*!*/;
# at 1021
#091018 18:35:42 server id 1 end_log_pos 1048 Xid = 1433
COMMIT/*!*/;

```

在示例 B-4 中，事务 trans\_id 245 是 slave-1 所见的最后一个事务，所以 slave-1 在文件 *slave-3-bin.000005* 的开始位置是字节位置 1048。所以，为了在正确的位置启动 slave-1，执行 CHANGE MASTER 和 START SLAVE：

```

slave-1> CHANGE MASTER TO
-> MASTER_HOST = 'slave-3',
-> MASTER_LOG_FILE = 'slave-3-bin.000005',
-> MASTER_LOG_POS = 1048;
Query OK, 0 rows affected (0.04 sec)

slave-1> START SLAVE;
Query OK, 0 rows affected (0.17 sec)

```

以这种方式将 slave 一个一个地连接到新 master 上的正确位置。

如果每个事务提交都添加了更新语句，这种方法很好。但是，有些语句在语句的前面或后面执行了隐式提交。典型的例子像 CREATE TABLE、DROP TABLE、ALTER TABLE。这个语句做隐式提交，所以它们无法被正确地打上标签，因此也就不能在语句刚结束的时候重新启动。假设执行了下面一组语句，然后出现崩溃：

```

INSERT INTO message_board VALUES ('mats@sun.com', 'Hello World!');
CREATE TABLE admin_table (a INT UNSIGNED);
INSERT INTO message_board VALUES ('', '');

```

如果 slave 刚刚执行完 CREATE TABLE 就失去了 master 连接，那么最后可见的全局事务标识符就是 CREATE TABLE 语句前面的 INSERT INTO。所以，slave 会使用 INSERT INTO 语句的事务标识符试图重新连接提升 slave。由于这个位置在提升 slave 的二进制日志中存

在，就会再次复制 CREATE TABLE 语句，导致 slave 出错并停止。

701 要避免这些问题，需要谨慎地设计和使用那些包含隐式提交的语句。例如，如果用 CREATE TABLE IF NOT EXISTS 代替 CREATE TABLE，slave 就会注意到表已经存在了，就跳过这个语句的执行。

第一步是远程获取 binlog 文件，与第 3 章中介绍的方法类似。这里，需要获取整个 binlog 文件，因为我们并不知道从哪里开始读取。示例 B-5 中的 fetch\_remote\_binlog 函数从服务器获取二进制日志，并返回二进制日志的行迭代器。

示例B-5：获取远程二进制日志

```
import subprocess

def fetch_remote_binlog(server, binlog_file):
    command = ["mysqlbinlog",
               "--read-from-remote-server",
               "--force",
               "--host=%s" % (server.host),
               "--user=%s" % (server.sql_user.name)]
    if server.sql_user.passwd:
        command.append("--password=%s" % (server.sql_user.passwd))
    command.append(binlog_file)
    return iter(subprocess.Popen(command, stdout=subprocess.PIPE).stdout)
```

迭代器一个一个地返回二进制日志的行，所以需要把这些行进一步分配到事务和事件中，以简化二进制日志的处理。在示例 B-6 中，group\_by\_event 函数将属于同一个事件的分到单个字符串中，group\_by\_trans 函数将事件流（由 group\_by\_event）分组到列表中去，每一个列表表示一个事务。

示例B-6：解析mysqlbinlog的输出以抽取事务

```
delimiter = "/*!*/;"

def group_by_event(lines):
    event_lines = []
    for line in lines:
        if line.startswith('#'):
            if line.startswith("# End of log file"):
                del event_lines[-1]
                yield ''.join(event_lines)
                return
            if line.startswith("# at"):
                yield ''.join(event_lines)
                event_lines = []
```



```

        event_lines.append(line)

def group_by_trans(lines):
    group = []

    in_transaction = False

    for event in group_by_event(lines):
        group.append(event)
        if event.find(delimiter + "\nBEGIN\n" + delimiter) >= 0:
            in_transaction = True
        elif not in_transaction:
            yield group
            group = []
        else:
            p = event.find("\nCOMMIT")
            if p >= 0 and (event.startswith(delimiter, p+7)
                          or event.startswith(delimiter, p+8)):
                yield group
                group = []
            in_transaction = False

```

最后一步是实现扫描服务器的二进制日志，并定位给定 GTID 的位置。从最近的二进制文件开始依次扫描更旧的文件。按照这个顺序扫描文件是因为假定函数的目的是 slave 提升，并且最近的二进制日志应该包含指定的 GTID，见示例 B-7。

示例B-7：从GTID中寻找文件位置

```

_GIDCRE = re.compile(r"^UPDATE Last_Exec_Trans SET\s+"
                    r"server_id = (?P<server_id>\d+)\s+"
                    r"trans_id = (?P<trans_id>\d+)\s+", re.MULTILINE)
_HEADCRE = re.compile(r"#\d{6}\s+\d?\d:\d\d:\d\d\s+"
                    r"server_id\s+(?P<sid>\d+)\s+"
                    r"end_log_pos\s+(?P<end_pos>\d+)\s+"
                    r"(?P<type>\w+)")

def scan_logfile(master, logfile, gtid): ❶
    from mysql.replicant.server import Position
    lines = fetch_remote_binlog(master, logfile)
    # 扫描输出以发现 GTID 的更新语句
    for trans in group_by_trans(lines):
        if len(trans) < 3:
            continue
        # 检查 Last_Exec_Trans 表的更新
        m = _GIDCRE.search(trans[-2])
        if m:

```

```

server_id = int(m.group("server_id"))
trans_id = int(m.group("trans_id"))
if server_id == gtid.server_id and trans_id == gtid.trans_id:
    # 检查 end_log_pos 的注释信息。假定只有 InnoDB 表，
    # 所以认为事务以一个 Xid 事件结束。
    m = _HEADCRCRE.search(trans[-1])
    if m and m.group("type") == "Xid":
        return Position(server_id, logfile, int(m.group("end_pos")))

return None

def find_position_from_gtid(server, gtid): ❷
    # 读 master 上的日志文件
    server.connect()
    logs = [ row["Log_name"] for row in server.sql("SHOW MASTER LOGS") ]
    server.disconnect()

    logs.reverse()
    for log in logs:
        pos = scan_logfile(server, log, gtid)
        if pos:
            return pos
    return None

```

- ❶ 这个函数扫描 *mysqlbinlog* 的输出结果寻找 GTID。参数包括：从哪个服务器获取 binlog 文件，需要扫描的 binlog 文件名（即服务器上的二进制日志文件名），以及需要查找的 GTID。返回含有这个 GTID 的事务的结束位置，或者如果在 binlog 文件中找不到这个 GTID，则返回 None。
- ❷ 这个函数以相反的顺序依次处理所有 binlog 文件。从最近的 binlog 文件开始，继续处理更旧的文件，直到在某个文件中找到 GTID 或者所有 binlog 文件都处理完毕。如果找到 GTID，则返回含有这个 GTID 的事务的结束位置。否则，返回 None。

## Symbols

- # (hash mark), 108
- \* (asterisk), 110
- log-slow-queries startup option, 408
- \_ (underscore), 175

## A

- acceptable downtime, 315
- active binlog file
  - described, 33, 53
  - switching, 102
- Activity Monitor, 361–365
- administrative tasks
  - adding slaves, 617–618
  - checking slave status, 612–615
  - load balancing functions, 165
  - managing replication, 11–15, 160
  - MySQL Workbench, 392–400
  - performing common, 42–49
  - stopping replication, 615
- advisors
  - described, 585, 594
  - MEM support, 594–595
- after image (column bitmaps), 282
- agents
  - described, 585
  - fixing problems, 588–590
  - installing, 587
- ALTER statement
  - binlog events and, 53
  - logging transactions, 86
- ALTER TABLE statement
  - changing tablespaces, 450
  - hierarchal replication, 172
  - logging, 59
  - MySQL Cluster and, 305
  - ORDER BY clause, 470
  - transactional replication and, 272
  - troubleshooting, 465
  - usage example, 438
- ANALYZE TABLE statement
  - best practices, 443
  - described, 431–434
  - repairing tables, 468
- Append\_block event, 65
- apply-incremental-backup command, 550
- apply-log command, 550
- archival plans
  - defined, 546
  - forming, 546–547
- Archive storage engine
  - described, 440
  - repairing tables, 468
- asterisk (\*), 110
- asynchronous replication
  - MySQL support, 6
  - value of, 156–158

†: 索引所列页码为本书英文版页码, 请参照正文侧边用“□”表示的原书页码。

autoextend option, 450

automating

- backups, 579–581

- monitoring with cron, 356

AUTO\_INCREMENT attribute

- bidirectional replication, 143

- context events, 62

- data sharding and, 193

- loading multiple tables, 99

- logging queries, 60

- usage example, 252

auto\_increment\_increment option, 143, 518

auto\_increment\_offset option, 143, 518

## B

backup command, 549

backup-and-apply-log command, 549–554

--backup-dir option, 554

backup-dir-to-image command, 550

backup-to-image command, 550

backups, 546

- (see also specific backup tools)

- automating, 579–581

- cloning slaves and, 39

- cloning the master and, 36

- data sharding and, 203, 222

- expectations for, 544

- FLUSH TABLES WITH READ LOCK state -  
ment and, 40

- forming archival plans, 546

- high availability and, 295

- importance of, 5, 542–544

- InnoDB tables, 40

- logical versus physical, 545

- LVM support, 564–568

- method comparisons, 569

- MySQL Cluster and, 304

- MySQL Enterprise Backup, 36, 41, 94, 548–  
559

- mysqldump utility, 560–562

- online methods, 129

- physical file copy, 562–564

- PITR and, 571

- procedure overview, 575

- replication and, 571

- scaling out and, 155

- schema, 225

- snapshots and, 566

- system availability and, 7

- terminology, 542

- XtraBackup, 569

bandwidth, replication monitoring and, 478,  
495

baselines, forming, 335

before image (column bitmaps), 282

BEGIN statement, 86, 280

Begin\_load\_query event, 65

benchmark function, 406, 444

benchmarking, 405–407

best practices

- checking configuration, 524

- checking logs, 523

- checking server status, 523

- common procedures, 526–528

- connection pools, 327

- constraint usage, 442

- data nodes, 327

- database optimization, 435–443

- hardware, 328

- high performance, 326

- for improving performance, 444–446

- indexes, 435, 442

- memory, 328

- nontransactional tables, 526

- normalization, 435

- operating systems, 328

- orderly restarts, 525

- orderly shutdowns, 525

- queries, 327, 444, 526

- query cache, 441

- replication, 445

- storage engines, 436–441

- topologies, 521–523

- transactional tables, 526

- troubleshooting, 521–528

- views, 441

bidirectional replication, 140–144, 190

binary log, 105

- (see also logging statements; mysqlbinlog  
utility)

- binary log group commit, 94–97

- cloning masters and, 37

- cloning slaves and, 39

- content considerations, 33–35, 52

- crash safety and, 100

- described, 25, 29, 52, 571

- execution order and, 249

- filtering, 67–69



- GTIDs and, 261
  - logging statements, 58–86
  - logging transactions, 86–97
  - maintaining replication positions, 246–248
  - managing, 100–105
  - mysqlbinlog utility and, 105–117
  - options and variables, 118–120
  - recording changes, 30
  - replication example, 30–32
  - replication threads, 481
  - row-based replication, 97–100
  - security and, 73
  - structure considerations, 33–35, 52–57
  - troubleshooting, 506
  - verbose option, 675
  - binary log events
    - checking, 31–32
    - described, 33, 52
    - event checksums, 56
    - formats for, 54–56
    - structure considerations, 52–54
    - troubleshooting, 503–505
  - binary log group commit, 94–97
  - binlog files
    - creating, 106
    - creation time, 117
    - described, 53
    - dumping contents, 106, 481
    - file rotation, 101–103
    - format version, 55, 102
    - GTID example, 266
    - processing order, 110
    - purging, 103, 104
    - reading, 112
    - shared disks and, 137
  - binlog index file, 53
  - binlog injector thread, 321
  - binlog-cache-size variable, 102, 119
  - binlog-checksum option, 57
  - binlog-do-db option
    - binary log filters, 67–69
    - creating master filters, 175
    - data sharding and, 225
    - filtering data, 479
    - server logs and, 409
  - binlog-format option, 99, 120, 321
  - binlog-ignore-db option
    - binary log filters, 67–69
    - creating master filters, 175
    - filtering data, 479
    - server logs and, 409
  - binlog-in-use flag, 93, 102, 115
  - binlog-max-row-event-size option, 120
  - binlog-row-event-max-size option, 280
  - binlog-row-image option, 288
  - binlog-rows-query-log-events option, 120
  - binlog\_direct\_non\_transactional\_updates option, 90
  - binlog\_max\_flush\_queue\_time variable, 97
  - binlog\_order\_commits variable, 96
  - Binomial tail distribution, 125
  - Birthday Problem, 194
  - Blackhole storage engine, 171–172, 439
  - bootstrapping a slave, 36
  - buffer pools
    - described, 449
    - InnoDB storage engine and, 458–460
    - monitoring, 458–460
  - Bunce, Tim, 564
  - business continuity (see information integrity)
- ## C
- CA (certification authority), 236–238
  - CACHE INDEX statement, 473
  - caches
    - buffer pools and, 449
    - key, 467, 471–474
    - load balancing, 167–168
    - transaction, 87–91
  - Cacti tool, 340
  - CALL statement, 77
  - candidate slaves, 651
  - central store
    - data sharding and, 192, 206
    - duplicating, 163
  - certification authority (CA), 236–238
  - chain topology, 522
  - CHANGE MASTER statement
    - best practices, 522
    - configuring master heartbeat, 496
    - configuring SSL connection, 495
    - connecting to downstream server, 150
    - Final role and, 20
    - hot standby and, 133
    - IGNORE\_SERVER\_IDS option, 518
    - master status variables and, 487
    - MASTER\_AUTO\_POSITION option, 618
    - MASTER\_HOST option, 235

- noting slave position, 137
- replication tips and tricks, 688
- round-robin multisource replication and, 277
- troubleshooting replication, 520
- troubleshooting slaves, 510, 512, 516
- CHANGE MASTER TO statement
  - configuring replication, 231
  - configuring slaves, 38
  - connecting master and slave, 28
  - CONNECT\_RETRY parameter, 248
  - MASTER\_AUTO\_POSITION option, 262–263
  - MASTER\_LOG\_FILE option, 36
  - MASTER\_LOG\_POS option, 36
  - MASTER\_SSL option, 237
  - MASTER\_SSL\_CAPATH option, 237
  - MASTER\_SSL\_CERT option, 237
  - MASTER\_SSL\_KEY option, 237
  - privilege considerations, 28
  - relay log information file, 232
  - replication status information, 241
  - slave promotion, 146
- channels, defined, 322
- checkpointing, 295, 316
- checksums
  - backup utilities and, 550
  - described, 340
  - event, 56
- circular replication
  - depicted, 158
  - described, 149–151
  - ignoring servers, 684
  - troubleshooting, 518
- circular topology, 523
- cloning operation
  - cloning users, 642
  - master servers, 37–38
  - scripting, 41–42
  - server clone utility and, 639–641
  - slave servers, 39–40
- cloud computing, MEM and, 607
- clustered indexes, 448
- column bitmaps, 279, 282
- command modules, 663
- comments, interpreting, 108
- Commit mutex, 95
- COMMIT statement
  - distributed transaction processing and, 91
  - logging transactions, 86
  - query events and, 280
  - two-phase commit and, 156
- commit\_and\_sync function, 179, 184
- common modules, 664
- composite identifiers, 194
- compress option, 554
- compressing tables, 462, 468, 471
- Com\_change\_master status variable, 487
- Com\_show\_master\_status status variable, 487
- Com\_show\_slave\_hosts status variable, 492
- Com\_show\_slave\_status status variable, 492
- Com\_slave\_start status variable, 492
- Com\_slave\_stop status variable, 492
- CONCAT function, 406
- configuration file
  - best practices, 524
  - binary log and, 53
  - binary log filters, 67–69
  - configuring masters, 25–26
  - configuring slaves, 27
  - data sharding and, 221
  - filtering options, 175–177
  - managing replication, 13
  - maximum allowed packet size, 66
  - MySQL Native Driver, 169
  - ndb-nodeid parameter, 310
  - relay servers, 171
  - semisynchronous replication setup, 258
  - Server class and, 17
  - SSL considerations, 237
  - starting SQL nodes, 311
  - troubleshooting replication, 520
- connection pools, 327
- CONNECTION\_ID function
  - row-based replication and, 97
  - session-specific, 249
  - thread ID and, 64
- consistency (see data consistency)
- Console application, 359–361, 466
- console option, 408, 454, 466
- constraints, best practices, 442
- context events
  - logging queries, 59–64
  - SQL threads and, 251–253
- contingency plans (high availability)
  - described, 124
  - disaster recovery, 127, 539
  - master failures, 127

- planning considerations, 126
- relay failures, 127
- slave failures, 127
- copy-back command, 550
- copy-back-and-apply-log command, 550
- CPU-bound processes, 337
- Create Basic Task Wizard, 48
- CREATE statement
  - binlog events and, 53
  - described, 468
  - ENGINE parameter, 438
  - logging transactions, 86
- CREATE FUNCTION statement, 78
- CREATE INDEX statement, 305
- CREATE PROCEDURE statement, 77
- CREATE ROUTINE privilege, 80–81, 119
- CREATE SCHEMA statement, 190
- CREATE TABLE statement, 59, 474
- CREATE TABLE IF NOT EXISTS statement, 52
- CREATE TEMPORARY TABLE statement, 498
- CREATE TRIGGER statement
  - DEFINER clause, 74
  - security considerations, 72
- CREATE USER privilege, 27
- Create\_file\_log\_event, 66
- cron facility, 342, 356
- crontab files, 48
- cross-schema queries, 202
- cross-shard joins, 193
- cryptography (see encryption)
- CSV storage engine
  - described, 439
  - repairing tables, 468
- CURDATE function, 61
- current database
  - binary log filters, 67, 679
  - logging queries, 59–61
  - query events and, 116
  - USE statement, 69
- current master, 277
- current time, logging queries, 60
- CURRENT\_ID function, 60
- CURRENT\_USER function, 97, 99
- CURTIME function, 61
- cycle function, 278
- Cygwin environment, 563

## D

- Dashboard
  - Advisors tab, 604
  - configuring, 590
  - consolidated server graphs, 601
  - critical issues details, 600–601
  - described, 585
  - MEM support, 591–594
  - Query Analyzer and, 596
  - Replication tab, 603
  - server details, 602
- Data Collector Sets, 376
- data consistency
  - asynchronous replication and, 156, 158
  - comparing databases for, 621–623
  - in hierarchal deployment, 180
  - managing, 177
  - MyISAM considerations, 100
  - in nonhierarchal deployment, 178–180
- Data Definition Language statements (see DDL statements)
- data dictionaries, 465
- data loss
  - common causes of, 543
  - troubleshooting, 514
- Data Manipulation Language (see DML)
- data nodes
  - adding, 305
  - best practices, 327
  - described, 292
  - IP addresses and, 309
  - node ID, 318
  - node recovery and, 318
  - starting, 310
- data protection, 532
- data recovery, 571
  - (see also disaster recovery)
  - backup and restore, 542–547
  - importance of, 541
  - PITR and, 571
  - recovery example, 572
  - recovery images, 573
  - replication and, 571
  - terminology, 542
- data sharding
  - backups and, 203, 222
  - common usage, 191–192
  - described, 6, 190
  - dispatching transactions, 195, 215–218

- dynamic, 206, 208
- elements of a solution, 194–196
- high-level architecture, 196
- limitations of, 192–194
- managing, 195, 220–225
- mapping sharding key, 194, 206–215
- partitioning data, 190, 194, 197–202, 210–215
- performance considerations, 191
- processing queries, 193, 195, 202, 215, 218–220
- shard allocation, 202–205
- slave lag and, 497
- static, 206–208
- virtual shards, 202–205
- writing data and, 156
- data-mining queries, 43
- databases
  - assigning to different masters, 141–144
  - comparing for consistency, 621–623
  - copying, 624–625
  - copying files, 129
  - corrupted, 466
  - cross-shard joins, 193
  - discovering differences in, 629–632
  - exporting, 625–628
  - importing, 628
  - measuring performance, 423–435
  - optimizing, 435–443, 478
  - performance information, 423
  - replication after crashes, 268–270
  - schemas and, 190
  - sharding, 192, 196, 198–202, 206
  - troubleshooting, 466
- datadir startup option, 467, 559, 567
- Date, C. J., 423
- DDL (Data Definition Language) statements
  - backups and, 561
  - described, 120
  - GTIDs and, 266
  - logging statements, 59
  - logging transactions, 86
- deadlocks, 465
- DEFAULT option, 443
- default-storage-engine option, 171
- defaults-file option, 558
- DEFINER clause
  - CREATE TRIGGER statement, 74
  - events, 81
  - stored functions, 78
  - stored procedures, 77, 117
  - stored routines, 75
  - triggers, 117
- defragmenting tables, 471
- delayed slaves, 6
- DELETE statement
  - data sharding and, 225
  - invoking triggers, 72
  - LIMIT clause, 97, 275, 287
  - logging, 58
  - nontransactional changes and, 275
  - stored procedures and, 76
  - WHERE clause, 52, 275
- Delete\_rows event, 279, 283
- deployment
  - described, 4, 12
  - hierarchal, 180
  - nonhierarchal, 178–180
- DESCRIBE statement, 424
- df command, 353
- diamond configuration, 276
- differential backups, 545
- disaster avoidance
  - hot standby, 154
  - remote replication and, 154
- disaster recovery, 535
  - (see also data recovery)
  - commands supported, 550
  - contingency plans, 127
  - goal of, 537
  - high availability and, 317, 541
  - information integrity and, 535–541
  - MySQL Cluster and, 295
  - planning considerations, 537
  - practicing, 539
  - slave servers, 268–275
  - tools and strategies, 540–541
  - workflow considerations, 538–539
- disk usage
  - filesystems and, 339
  - monitoring, 336, 338, 350–353, 363
  - MyISAM storage engine and, 467
  - optimizing, 467
  - showing, 632–635
- Disk Usage Analyzer, 352
- disk-bound processes, 338
- distributed data, redundancy and, 297



- Distributed Replicated Block Device (DRBD), 138
- distributed transaction processing, 91–94
- distribution costs, managing replication, 160
- dmesg utility, 349
- DML (Data Manipulation Language)
  - GTIDs and, 266
  - logging statements, 58
  - row-based replication and, 120
  - troubleshooting tips, 517
- DRBD (Distributed Replicated Block Device), 138
- DROP statement, 440
- DROP INDEX statement, 305
- DROP TABLE statement, 35, 465
- DROP TABLE IF EXISTS statement, 52
- dual-master setup
  - active-active setup, 135–136, 140–144
  - active-passive setup, 135–136, 139
  - bidirectional replication, 140–144
  - depicted, 158
  - described, 6
  - high availability and, 135–144
  - managing replication, 12
  - replicated disks and, 138
  - shared disks and, 137–138
- DuBois, Paul, 8
- dump threads, 228
- dynamic sharding, 206, 208

## E

- EmptyRowError exception, 15
- encryption
  - hash functions and, 213
  - security considerations, 73
  - SSL support, 235–237
- enforce\_gtid\_consistency variable, 498
- epochs (transactions), 297
- Error class, 15
- error handling
  - console messages and, 466
  - exceptions and, 15–16
  - HA\_ERR\_KEY\_NOT\_FOUND error, 519
  - log messages and, 371
  - logging statements, 83–86
  - troubleshooting, 517
- Event Viewer, 369–372
- events
  - affecting replication, 100

- binary log, 31–32, 33, 52, 54–56
- checksums for, 56
- DEFINER clause, 81
- described, 81
- failover, 658–662
- filtering, 174–176, 255–256, 419
- incidents and, 102, 103
- interpreting, 113–117
- logging statements, 70–75, 81
- monitoring, 411
- partitioning to slaves, 176
- password considerations, 73
- row-based replication, 278–286
- setting timestamps for, 108
- skipping, 251, 255–256
- slave processing, 249–256
- SQL thread processing, 250–256
- triggers and, 284–286
- excessive lag, 513
- Execute\_load\_query event, 65
- Execute\_log\_event, 66
- expire-logs-days option, 104, 118
- EXPLAIN statement
  - best practices, 443
  - described, 423–431
  - executing queries, 596
  - indexes and, 435
  - usage example, 401
- EXTENDED keyword, 427
- extended-status command, 390
- external replication, 319
- extract command
  - backup-dir option, 550
  - src-entry option, 550

## F

- failover
  - described, 315, 651
  - GTIDs handling, 131, 263–264
  - high availability and, 314
  - multichannel replication and, 323, 678
  - MySQL Cluster and, 295
  - MySQL Utilities and, 655–663
  - replication and, 478
- failover event, 658–662
- Faroult, Stephane, 423
- fault tolerance, 316
- fdisk command, 353
- Federated storage engine, 440

- fetch\_master\_pos function, 41, 179, 184
- fetch\_relay\_chain function, 185
- fetch\_remote\_binlog function, 46
- fetch\_slave\_pos function, 41
- fetch\_trans\_id function, 184
- file IDs, 66
- filesystem
  - coordinating synchronization, 101
  - DDL statements and, 59
  - disk usage and, 339
  - logging changes, 86
  - logical volumes and, 566
  - memory considerations, 338
  - reading remote files, 111
  - snapshot support, 129
- filtering
  - binary log filters and, 67–69
  - current database, 679
  - events, 174–176, 255–256, 419
  - inclusive and exclusive replication and, 479
  - partitioning events to slaves, 176
  - in row-based replication, 286–287
  - scaling out and, 155
  - SQL threads and, 250
- Final role, 20
- find\_datetime\_position function, 46
- FLUSH LOGS statement
  - binlog file support, 53
  - described, 102
  - monitoring master servers, 484
  - privilege considerations, 28
  - troubleshooting binary log, 506
  - usage example, 31
- Flush mutex, 95
- FLUSH QUERY CACHE statement, 389
- FLUSH TABLES statement, 563
- FLUSH TABLES WITH READ LOCK state -  
ment
  - cloning the master, 37
  - LVM support, 567
  - making backups and, 40
  - pausing replication, 527
  - releasing locks, 18
- foreign keys, 443
- format description events
  - in binary log, 102
  - binlog event structure, 53, 55
  - described, 32–33
  - I/O threads and, 250
  - post header and body, 117
  - printing, 108
  - XA and, 93
- FOUND\_ROWS function, 507
- fragments, data, 320
- free command, 341, 347
- .frm file extension, 465, 468
- fsync call, 95, 269, 458
- FULL keyword, 383
- full-instance backups, 549
- functional partitioning (see data sharding)

## G

- general-log startup option, 408
- get\_connection function, 165
- GLOBAL keyword, 384
- global redundancy, 296, 324
- global tables
  - described, 192
  - sharded example, 198–202
- global transaction identifier set (GTID set), 260
- global transaction identifiers (see GTIDs)
- Gnome System Monitor, 341
- GRANT OPTION privilege, 28
- graphical user interfaces (GUIs), 357, 391
- grep command, 346
- groups
  - described, 53
  - group commit, 94–97
  - volume, 565
- GTID class, 16
- GTID set (global transaction identifier set), 260
- GTIDs (global transaction identifiers)
  - adding to servers, 693–697
  - described, 36, 260, 650
  - finding positions of, 699–703
  - handling failover, 131, 263–264
  - managing data consistency, 178, 181, 184, 187
  - monitoring slaves, 490
  - MySQL Workbench and, 494
  - replication and, 266–267, 498–499, 519
  - setting up replication using, 261–262
  - slave promotion, 148, 264
  - transaction handling using, 697
- gtid\_executed variable, 264, 266, 498, 519
- gtid\_mode variable, 499
- gtid\_next variable, 267, 499
- gtid\_owned variable, 499

gtid\_ownedl variable, 267  
gtid\_purged variable, 264, 499, 519  
GUIs (graphical user interfaces), 357, 391

## H

### hardware

- best practices, 328
- monitoring, 335–340
- node recovery and, 318
- real-world failures, 543

### hash mapping (data sharding)

- adding new shards, 214
- consistent hashing and, 212–213
- creating index tables, 214
- described, 210, 212
- fetching shards, 215

### hash mark (#), 108

### HA\_ERR\_KEY\_NOT\_FOUND error, 519

### health command, 615

### Health Insurance Portability Accountability Act (HIPAA), 534

### heartbeat mechanism

- configuring for masters, 496
- described, 495
- monitoring slaves and, 492
- multichannel replication and, 322
- node recovery and, 318
- replication tips and tricks, 683

### hierarchal deployment, 180

### hierarchal replication

- described, 170
- setting up relay servers, 171–172

### high availability

- achieving, 314–316
- backups and, 7
- contingency plans, 124, 126–128
- disaster recovery and, 541
- hot standby and, 23
- information integrity and, 534
- MyISAM storage engine and, 475
- node recovery and, 318
- procedures and, 124, 128–151
- redundancy and, 6, 124–126, 296, 324
- replication and, 319–324
- system recovery and, 317
- utilities supporting, 650–663

### HIPAA (Health Insurance Portability Accountability Act), 534

### hit ratio, 459

### horizontal partitioning (see data sharding)

### host-bin option, 118

### hostname-bin option, 26

### hostnames

- connecting master and slave, 28
- MEM installation and, 588

### hot standby

- described, 23, 130–131
- disaster avoidance through, 154
- switching over with GTIDs, 263

### hybrid topology, 523

## I

### I/O threads

- described, 233
- starting and stopping, 277, 615
- troubleshooting, 515

### ifconfig command, 354

### IGNORE LEAVES clause, 473

### image-to-backup-dir command, 550

### incident events, 102, 103

### incremental backups, 545, 550, 554–557

### --incremental-base option, 554

### indexes

- best practices, 435, 442
- CACHE INDEX statement, 473
- checking table, 635
- clustered, 448
- preloading into key caches, 472
- queries and, 387
- storing tables in index order, 470

### information assurance

- described, 532
- importance of, 533
- related practices, 532

### information integrity

- backup and restore, 542–547
- data recovery and, 541
- described, 532–534
- disaster recovery and, 535–541
- high availability and, 534

### information significance, 532

### INFORMATION\_SCHEMA database tables, 383, 461–462

### init-file command, 474

### init\_file option, 503

### InnoDB Hot Backup application, 569

- InnoDB monitors
  - SHOW ENGINE INNODB STATUS state -  
ment, 450–453
  - troubleshooting with, 465
  - usage overview, 453–457
- InnoDB storage engine
  - architectural features, 448–457
  - backing up tables, 40
  - described, 438
  - dual-master setup and, 137
  - FLUSH TABLES WITH READ LOCK state -  
ment and, 40
  - improving performance, 448
  - incremental backups and, 557
  - INFORMATION\_SCHEMA database tables,  
461–462
  - InnoDB system activity report, 404
  - monitor mechanism, 453–457
  - monitoring buffer pools, 458–460
  - monitoring logfiles, 457
  - monitoring tablespaces, 460
  - parameters supported, 463
  - Performance Schema feature and, 462–464
  - recovery considerations, 139
  - SHOW ENGINE INNODB STATUS state -  
ment, 450–453
  - slave lag and lock contention, 498
  - snapshot support, 129
  - transactional replication and, 272
  - troubleshooting, 464–467
  - XA support, 101
- InnoDB\_buffer\_pool\_pages\_data status vari -  
able, 459
- InnoDB\_buffer\_pool\_pages\_dirty status vari -  
able, 459
- InnoDB\_buffer\_pool\_pages\_flushed status vari -  
able, 460
- InnoDB\_buffer\_pool\_pages\_free status variable,  
460
- InnoDB\_buffer\_pool\_pages\_misc status vari -  
able, 460
- InnoDB\_buffer\_pool\_pages\_total status vari -  
able, 460
- InnoDB\_buffer\_pool\_reads status variable, 460
- InnoDB\_buffer\_pool\_read\_ahead\_rnd status  
variable, 460
- InnoDB\_buffer\_pool\_read\_ahead\_seq status  
variable, 460
- InnoDB\_buffer\_pool\_read\_requests status vari -  
able, 460
- InnoDB\_buffer\_pool\_wait\_free status variable,  
460
- InnoDB\_buffer\_pool\_write\_requests status  
variable, 460
- innodb\_fast\_shutdown option, 464
- innodb\_file\_per\_table option, 450
- innodb\_file\_per\_table option, 465, 548
- innodb\_force\_recovery recovery option, 466
- innodb\_lock\_wait\_timeout variable, 464
- InnoDB\_log\_waits status variable, 457
- InnoDB\_log\_writes status variable, 457
- InnoDB\_log\_write\_requests status variable, 457
- InnoDB\_os\_log\_fsyncs status variable, 458
- InnoDB\_os\_log\_pending\_fsyncs status variable,  
458
- InnoDB\_os\_log\_pending\_writes status variable,  
458
- InnoDB\_os\_log\_written status variable, 458
- innodb\_print\_all\_deadlocks option, 465
- innodb\_thread\_concurrency option, 463
- innodb\_undo\_directory option, 450
- innodb\_undo\_tablespaces option, 449
- InnoDB system activity report, 404
- INSERT statement
  - best practices, 326
  - invoking triggers, 72, 75
  - LIMIT clause, 97, 287
  - logging, 58
  - nontransactional changes and, 84, 275
  - stored functions and, 79
  - stored procedures and, 76
  - usage example, 108
  - usage examples, 253
- INSERT DELAYED statement, 507
- INSERT INTO statement, 511
- INSERT\_ID session variable, 108, 251
- integer data, interpreting, 114
- internal replication, 319
- Internet, running replication over, 235–239
- Intvar event
  - described, 62, 251
  - mysqlbinlog support, 108, 110
  - stored procedures and, 78
- inventory assessment, 539
- I/O threads
  - handling broken connections, 248
  - housekeeping, 249



- replication and, 481
- starting and stopping, 234
- state considerations, 243–246
- synchronizing, 269
- I/O transfer rates, 351
- ionice command, 343
- iostat command, 341, 344, 350
- IP addresses
  - connecting master and slave, 29
  - data nodes and, 309
- ISO-3309 standard, 57
- itertools module, 278

## K

- KDE System Guard, 341
- kernel, memory and, 338
- key caches
  - creating, 473
  - described, 467
  - monitoring, 471
  - multiple, 473
  - preloading, 472
- KILL statement, 383, 390
- Kneschke, Jan, 161
- ksar tool, 403

## L

- LAST\_INSERT\_ID function, 60, 61, 251
- legal requirements, data preservation, 7
- LIKE clause, 384, 387
- LIMIT clause
  - DELETE statement, 97, 275, 287
  - INSERT statement, 97, 287
  - SELECT statement, 164
  - troubleshooting, 507
  - UPDATE statement, 97, 287
- Linux class, 17
- Linux High Availability project, 138
- Linux operating system
  - automated monitoring, 356
  - disk usage, 350–353
  - general system statistics, 355–356
  - LVM support, 36, 40, 129
  - memory usage, 347–349
  - monitoring, 334, 341–356
  - MySQL Replicant Library and, 12, 17
  - network activity, 353–355
  - process activity, 342–347

- Stunnel support, 239
- list mapping (data sharding), 210
- list-image command, 550
- load balancing
  - administrative tasks, 165
  - application-level, 162–170
  - caching, 167–168
  - for writes, 154
  - managing replication, 159
  - MySQL Native Driver and, 168–170
  - proxies and, 160
  - for reads, 154
  - slave lag and, 498
- LOAD DATA INFILE statement
  - handling current database, 60
  - LOAD\_FILE function and, 82
  - logging statements, 65–67
  - logging transactions, 87
- LOAD INDEX statement, 473
- LOAD\_FILE function, 82, 99
- Load\_log\_event, 66
- LOCAL keyword, 434
- local query handler, 304
- local recovery, 295
- local redundancy, 296, 324
- LOCK TABLES statement, 221
- locks
  - coordinating requests, 222
  - InnoDB lock monitor, 455, 465
  - releasing, 18
  - slave lag and, 498
  - table, 40
- LOCK\_commit mutex, 95
- LOCK\_commit\_queue mutex, 95
- LOCK\_flush\_queue mutex, 95
- LOCK\_log mutex, 58, 95
- LOCK\_sync mutex, 95
- LOCK\_sync\_queue mutex, 95
- log-bin option
  - controlling binlog files, 53
  - described, 25, 118, 409
  - Server class and, 17
  - slave promotion and, 145
- log-bin-index option
  - controlling binlog files, 53
  - described, 26, 118, 409
  - Server class and, 17
- log-bin-trust-function-creators option, 81, 119
- log-error startup option, 408, 464

- log-output startup option, 408
  - log-slave-updates option
    - bidirectional replication, 141
    - binary logging and, 510
    - GTIDs and, 261
    - hierarchal replication, 171
    - slave promotion and, 145
  - log-slow-slave-statements option, 408
  - logfiles
    - best practices, 523
    - Console application support, 359–361
    - Event Viewer support, 370
    - InnoDB storage engine and, 449, 457
    - monitoring, 457
    - RESET\_SLAVE statement and, 231
    - as separate tablespaces, 449
    - server logs, 407–409
    - troubleshooting data loss, 514
    - troubleshooting replication, 521
  - logging (MySQL Cluster), 295, 316
  - logging statements
    - binary log filters, 67–69
    - DDL statements, 59
    - DML statements, 58
    - error handling, 83–86
    - events, 70–75, 81
    - LOAD DATA INFILE statement, 65–67
    - LOCK\_log mutex, 58
    - nontransactional changes, 83–86
    - query events, 59–65
    - special constructions, 82
    - stored functions, 75, 78–81
    - stored procedures, 75–78
    - stored routines, 70–75
    - triggers, 70–75
  - logging transactions
    - binary log group commit, 94–97
    - implicit commits and, 86
    - starting transactions, 86–97
    - transaction cache, 87–91
    - XA support, 91–94
  - logical backups, 545
  - logical sequence number (LSN), 555
  - Logical Volume Manager (see LVM)
  - logical volumes, 565
  - Loukides, Mike, 351
  - ls command, 353
  - LSN (logical sequence number), 555
  - Lua programming language, 161
  - lvcreate command, 566
  - LVM (Logical Volume Manager)
    - backup and restore example, 567
    - backup comparisons, 569
    - cloning slaves, 40
    - cloning the master, 36
    - described, 564–568
    - getting started, 565–567
    - snapshot support, 129
  - lvremove command, 566
  - lvscan command, 566
  - L'Hermite, Pascal, 423
- ## M
- Mac OS X operating system
    - Activity Monitor, 361–365
    - Console application, 359–361
    - disk usage, 363
    - memory usage, 363
    - monitoring, 334, 356–365
    - MySQL Replicant Library and, 12
    - network activity, 364
    - System Profiler, 357–359
  - Machine class, 16
  - management buy-in, 538
  - managing binary log
    - binlog file rotation, 101–103
    - crash safety, 100
    - described, 100
    - incidents, 103
    - purging binlog file, 104
  - master dump thread, 233
  - master filters, 174–175
  - master log information file
    - described, 230
    - flushing, 269
    - fsync calls and, 269
    - manipulating slave threads, 234
    - replication status information, 246
    - storing replication information, 271
    - transactional replication and, 272
  - Master role
    - described, 20
    - replicate\_from function and, 41
  - master servers
    - assigning tables to different, 141–144
    - checking status, 523
    - circular replication, 149–151
    - cloning, 37–38

- configuring, 25–26
- connecting to slaves, 26–29
- creating, 7
- creating replication users, 26
- delayed slaves, 6
- dual-master setup, 6, 12, 135–144
- handling failures, 127, 130
- hierarchal replication, 170
- monitoring, 483–486
- monitoring thread status, 481
- multimaster issues, 518
- replication overview, 5
- scripting the clone operation, 41
- server roles, 19–20
- status variables and, 487
- switching, 130–134
- tips and tricks, 680–682
- troubleshooting, 503–509
- two-phase commit and, 156
- upgrading, 130
- master-connect-retry option, 248
- master-info-file option, 230
- master-retry-count option, 248
- master-verify-checksum option, 57
- master\_heartbeat\_period option, 496
- master\_info\_repository option, 271, 617
- MASTER\_POS\_WAIT function
  - data consistency example, 178–181
  - data sharding and, 221
  - described, 46
  - relay log processing and, 247
- max-allowed-packet option, 66, 515
- max-binlog-cache-size option, 119
- max-binlog-size option, 119
- MD5 function, 213, 253
- MEM (see MySQL Enterprise Monitor)
- Memcached technique, 163
- memory
  - best practices, 328
  - cautions tweaking, 338
  - monitoring, 336–338, 347–349, 363
  - node recovery and, 319
  - troubleshooting, 503, 513
- Memory storage engine, 439
- memory-bound processes, 338
- Merge storage engine, 440
- MERGE view, 142
- metadata search utility, 636
- mission statements, 538
- mixed-mode replication, 99
- Mollinaro, Anthony, 423
- monitoring, 381
  - (see also performance considerations)
  - automated, 356
  - benefits of, 335
  - buffer pools, 458–460
  - categories of, 334
  - described, 334
  - disk usage, 336, 338, 350–353, 363
  - events, 411
  - InnoDB storage engine, 448–467
  - key caches, 471
  - Linux operating system, 334, 341–356
  - logfiles, 457
  - Mac OS X operating system, 334, 356–365
  - master servers, 483–486
  - memory, 336–338, 347, 363
  - MyISAM storage engine, 467–475
  - MySQL Enterprise Monitor, 599–605
  - MySQL servers, 381–407
  - MySQL taxonomy, 421–422
  - MySQL Workbench and, 493–494
  - network activity, 336, 339, 353–355, 364
  - as preventive maintenance, 377
  - process activity, 342–347
  - processors, 335–337
  - replication, 477–500
  - semisynchronous replication, 259
  - slave lag, 496
  - slave servers, 487–492
  - statistics, 355–356, 606
  - tablespaces, 460
  - tools for, 340
  - Unix operating system, 334, 341–356
  - Windows operating system, 334, 365–377
- monitoring agents, 588–590, 594
- monitoring, usage examples, 7
- MONyog tool, 404
- mount command, 566
- mpstat command, 342, 344–346
- multichannel replication, 322–324, 678
- multimaster topology, 518, 523
- multisource replication, 275–278, 678, 687, 690
- Musumeci, Gian-Paolo D., 351
- mutexes
  - binary log group commit process and, 94
  - InnoDB storage engine information, 452
- mysam ftdump utility, 468

- MyISAM storage engine
  - compressing tables, 471
  - consistency considerations, 100
  - defragmenting tables, 471
  - described, 438
  - dual-master setup and, 137
  - high availability and, 475
  - improving performance, 467
  - maintaining row order, 287
  - monitoring key caches, 471
  - multiple key caches, 473
  - nontransactional changes and, 83, 86, 274
  - optimizing disk storage, 467
  - parameters supported, 474
  - preloading key cache, 472
  - query cache and, 387
  - recovery considerations, 139
  - repairing tables, 468
  - replication and, 475
  - slave lag and lock contention, 498
  - special utilities supported, 468–470
  - tables in index order, 470
  - transactional replication and, 272
  - troubleshooting tables, 514
- myisam-recover option, 506
- myisamchk utility
  - defragmenting tables, 471
  - described, 469–470
  - sort records option, 470
- myisamlog utility, 468
- myisampack utility, 468, 471
- myisam\_data\_pointer\_size option, 474
- myisam\_max\_sort\_file\_size option, 474
- myisam\_recover\_options option, 474
- myisam\_repair\_threads option, 475
- myisam\_sort\_buffer\_size option, 475
- myisam\_stats\_method option, 475
- myisam\_use\_mmap option, 475
- MySAR system activity report, 403
- MySQL
  - additional information, 8
  - monitoring taxonomy, 421–422
  - version considerations, 12
- MySQL Administrator application, 394
- MySQL Cluster
  - additional information, 305
  - architecture basics, 298–305, 321
  - best practices, 326
  - commit support, 157
  - data storage, 300–303
  - described, 292
  - example configuration, 306–314
  - features, 294–296
  - getting started, 306–308
  - high availability and, 314–324
  - high performance and, 324
  - incident events, 104
  - log handling, 297
  - management node, 308
  - NDB management console, 309
  - online operations, 304
  - partitioning and, 296, 303
  - redundancy and, 296–298, 324
  - replication and, 320, 678
  - shutting down clusters, 314
  - starting, 308–313
  - starting data nodes, 310
  - terminology and components, 292
  - testing clusters, 313
  - transaction management, 304
  - typical configuration, 293
- mysql database
  - logging transactions, 86
  - MySQL Cluster and, 321
- MySQL Enterprise Backup
  - backing up downed servers, 559
  - binary log group commit and, 94
  - cloning slaves, 39, 41
  - cloning the master, 36
  - commands supported, 549–550
  - features supported, 548–549
  - full backups, 551–554
  - incremental backups, 554–557
  - process overview, 551
  - restoring data, 557–559
  - snapshot support, 129
- MySQL Enterprise Monitor
  - additional information, 341, 608
  - advisors, 594–595
  - anatomy of, 585–586
  - clouding computing and, 607
  - commercial offerings, 585
  - Dashboard, 591–594
  - described, 322, 394, 584
  - installing, 586–588
  - key features, 590–597
  - monitoring agents, 594
  - monitoring areas supported, 599–605



- MySQL production support, 597
- Query Analyzer, 595, 605–608
- usage considerations, 597–608
- MySQL Monitor and Advisor (MONyog) tool, 404
- MySQL Native Driver (myslqnd), 168–170
- MySQL Protocol, 218
- MySQL Proxy
  - additional information, 161
  - load balancing and, 160
  - multimaster replication, 676
  - reporting statistics, 606
- MySQL Replicant Library
  - additional information, 8
  - automating steps with, 222
  - basic classes and functions, 15–16
  - described, 11–15
  - load balancing functions, 165
  - multisource replication and, 278
  - Server class, 17–19
  - server roles supported, 19–20
  - supporting different operating systems, 12, 16
- MySQL servers
  - benchmark suite, 405–407
  - communicating performance, 381
  - database performance, 423–443
  - key distribution and, 468
  - monitoring, 381–407
  - monitoring taxonomy, 421–422
  - MySQL Workbench, 391–402
  - mysqldadmin utility, 389–390
  - performance monitoring, 382
  - Performance Schema feature, 409–421
  - server logs, 407–409
  - showing server information, 641
  - SQL commands, 383–389
  - third-party tools, 402
- MySQL Utilities
  - architecture of, 663
  - checking replication setup, 646–648
  - checking table indexes, 635
  - cloning servers, 639–641
  - cloning users, 642
  - comparing databases for consistency, 621–623
  - copying databases, 624–625
  - custom utility example, 664–672
  - databases differences, 629–632
  - described, 559, 618
  - exporting databases, 625–628
  - getting started, 618
  - GTID problems and, 519
  - high availability utilities, 650–663
  - identifying differences in data and structure, 514
  - importing databases, 628
  - MySQL Workbench and, 559, 619–621
  - reading frm files, 468
  - searching for processes, 637–639
  - searching metadata, 636
  - setting up replication, 644
  - showing disk usage, 632–635
  - showing server information, 641
  - showing topologies, 648
  - thread inconsistency problems, 516
  - utilities client, 643
- MySQL Workbench
  - administration feature, 392–400
  - configuration tool, 397–398
  - data export and restore features, 399–400
  - described, 391
  - management group of tools, 393–397
  - monitoring slave servers, 487
  - MySQL Enterprise Backup and, 549
  - MySQL Utilities and, 559, 619–621
  - mysqldump utility and, 399
  - replication monitoring, 493–494
  - security tool, 398
  - SQL Editor and, 400–402
- mysqldadmin utility
  - commands supported, 389–390
  - MySQL Workbench and, 392
  - relative option, 390
  - sleep option, 390
- mysqlbinlog utility
  - read-from-remote-server option, 111
  - start-datetime option, 111
  - start-position option, 110
  - stop-position option, 111
  - base64-output=never option, 108
  - basic usage, 106–111
  - described, 105, 233, 386
  - force option, 44
  - force-if-open option, 107
  - GTID events and, 267
  - hexdump option, 113–114
  - interpreting comments, 108

- interpreting events, 113–117
- PITR and, 571
- pseudo\_thread\_id variable, 65
- reading raw binary log files, 112
- reading remote files, 111
- result-file option, 113
- short-form option, 107–108
- start-datetime option, 44
- stop-datetime option, 44, 111
- stop-never option, 113
- to-last-log option, 113
- troubleshooting replication, 521
- usage example, 45
- verbose option, 486
- verify-binlog-checksum option, 57
- viewing error codes, 86
- wildcard support, 110
- mysqldbcompare utility
  - comparing databases for consistency, 621–623
  - slave promotion, 146
  - thread inconsistency problems, 516
- mysqldbcopy utility
  - copying databases, 624–625
  - GTID problems and, 519
- mysqldbexport utility
  - described, 560
  - exporting databases, 625–628
  - GTID problems and, 519
- mysqldbimport utility
  - described, 560
  - GTID problems and, 519
  - importing databases, 628
- mysqldiff utility, 629–632
- mysqldiskusage utility, 632–635
- mysqldump utility
  - backup comparisons, 569
  - cloning slaves, 40
  - cloning the master, 36–38
  - described, 560–562
  - GTID problems and, 519
  - MySQL Workbench and, 399
  - options supported, 561
  - restoring data, 40
  - snapshots and, 129
- mysqlfailover utility, 655–663
- mysqlfrm utility, 468
- mysqlindexcheck utility, 635
- mysqlmetagrep utility, 636

- mysqlnd (MySQL Native Driver), 168–170
- mysqlprocrep utility, 637–639
- mysqlreplicate utility, 644
- mysqlrpladmin utility, 615, 651–655
- mysqlrplcheck utility, 646
- mysqlrplshow utility, 648
- mysqlserverclone utility, 639–641
- mysqlserverinfo utility, 641
- mysqluc wrapper, 643
- mysqluserclone utility, 642
- mytop utility, 403

## N

- Nagios tool, 341, 377
- NAME\_CONST function, 77
- NAT (network address translation), 245
- NDB (network database), 292, 298–300
- NDB management console
  - issuing SHUTDOWN statement, 314
  - snapshot backups, 304
  - starting, 309
  - system recovery, 318
- ndb-cluster-connection-pool option, 327
- ndbcluster option, 311
- NDBcluster storage engine, 313
- ndbd (NDB data node daemon)
  - described, 310
  - high availability example, 316
  - initial-start option, 310
  - ndb-connectstring option, 310
- ndb\_mgmd (NDB management daemon)
  - config-file option, 308
  - described, 299
  - example configuration, 306
  - high availability example, 316
  - initial option, 308
  - ndb-connectstring option, 310
  - rolling restart, 305
  - starting, 308
- ndb\_nodeid option, 312
- ndb\_restore utility, 305, 318
- netstat command, 342, 354
- network activity
  - improving replication performance, 495
  - monitoring, 336, 339, 353–355, 364
- network address translation (NAT), 245
- network database (NDB), 292, 298–300
- network-bound processes, 339, 340
- nice command, 343

- no-connection option, 559
- node ID, 318
- node recovery, 295, 318
- nonhierarchical deployment, 178–180
- nontransactional changes
  - avoiding problems with, 89
  - best practices, 526
  - data consistency problems, 100
  - error handling and, 83–86
  - implicit commits and, 86
  - logging, 88–89
  - protecting, 274
  - row-based replication and, 97
  - statement execution and, 88–89
  - troubleshooting, 507–509, 515
- NoOptionError exception, 15
- normalization, 435
- NOT NULL option, 443
- NotMasterError exception, 16
- NotSlaveError exception, 16
- NOW function, 60, 61
- NO\_WRITE\_TO\_BINLOG keyword, 434

## O

- ONLINE keyword, 305
- online operations, 296, 304
- operating systems, 340
  - (see also specific systems)
  - best practices, 328
  - class methods, 16
  - monitoring solutions, 340
  - MySQL Replicant Library and, 12
  - node recovery and, 319
- OPTIMIZE TABLE statement
  - best practices, 443
  - defragmenting tables, 471
  - described, 434
  - myisam\_recover\_options option and, 474
  - repairing tables, 468
- ORDER BY clause
  - ALTER TABLE statement and, 470
  - LIMIT clause and, 507
  - partial execution of statements and, 287
- ORDER BY RAND() clause, SELECT statement, 164
- overall transfer rate, 338

## P

- paging technique, 337
- partitioning
  - data sharding and, 190, 197–205, 210–215
  - described, 565
  - events to slaves, 176
  - MySQL Cluster and, 296, 303
- passwords
  - master log information file, 231
  - MEM installation and, 588
  - security considerations, 72
  - user account, 28
- Patriot Act, 534
- pausing replication, 527
- peak loads, handling, 159
- per-process transfer rate, 338
- Percona open source provider, 569
- performance considerations, 326
  - (see also monitoring)
  - best practices, 326, 444–446
  - data sharding and, 191
  - data-mining queries, 43
  - database, 423–443
  - database object manipulation, 59
  - described, 380
  - diagnosing performance problems, 420
  - high performance, 325–326
  - InnoDB storage engine, 448
  - MyISAM storage engine, 467
  - MySQL Cluster and, 324
  - MySQL servers, 381–419
  - optimizing views and, 143
  - Performance Monitor, 375–377
  - proxies and, 160
  - replication and, 477
  - report generation, 24, 155
  - synchronous replication, 157
- Performance Monitor, 375–377
- Performance Schema feature
  - described, 409–412
  - diagnosing performance problems, 420
  - getting started, 412
  - InnoDB storage engine and, 462–464
- performance-schema startup option, 412
- Perl programming language, 405
- PHP programming language, 162, 180
- physical backups, 545
- physical file copy, 562–564, 569
- physical volumes, 565

- pid-file option
    - described, 26, 118
    - Server class and, 17
  - PITR (point-in-time recovery)
    - backup in replication and, 571
    - backup procedure, 575
    - binary log and, 30, 59, 74, 176, 409
    - described, 5, 617
    - filtering considerations, 174
    - FLUSH LOGS statement and, 102
    - Python and, 575–579
    - recovery example, 572
    - recovery images, 573
    - replication and, 479
    - restoring after replicated error, 572
  - pmap command, 342, 348
  - polling, 183
  - pool\_add function, 165
  - pool\_del function, 165
  - pool\_set function, 165
  - port numbers
    - connecting master and slave, 28
    - Query Analyzer usage, 606
  - Position class, 16
  - post headers
    - described, 55
    - format description events, 117
    - query events, 115
  - postfiltering, 419
  - prefiltering, 419
  - primary servers, 136
  - primary-backup configuration, 131
  - private keys, 236
  - privileges
    - configuring replication, 26, 27
    - granting in production environment, 25
    - MySQL Workbench and, 398
    - reading remote files, 111
    - security and binary log, 73
    - setting thread IDs, 65
    - stored functions and, 80
  - proactive monitoring, 335
  - procedures (high availability)
    - best practices, 526–528
    - circular replication, 149–151
    - considerations for, 128–130
    - described, 124
    - dual-master setup, 135–144
    - hot standby, 130–134
    - semisynchronous replication, 136
    - slave promotion and, 130, 144–149
  - process IDs
    - temporary tables and, 64
    - TLS support, 254
  - processes
    - assigning priorities, 337, 343
    - CPU-bound, 337
    - described, 336
    - disk-bound, 338
    - killing runaway, 337
    - memory-bound, 338
    - monitoring activity, 342–347
    - network-bound, 339, 340
    - processor-bound, 337
    - removing unnecessary, 337
    - rescheduling, 337
    - searching for, 637–639
    - solutions to overloading, 336
    - spawning, 347
  - processlist command, 390
  - processor-bound processes, 337
  - processors, monitoring, 335–337
  - proxies
    - described, 160
    - distributing queries, 160
    - handling transactions, 217
    - loading balancing, 160
    - performance considerations, 160
  - ps command, 341, 346–347
  - pseudthread ID, 254
  - pseudo\_thread\_id server variable, 65
  - public certificates, 236
  - PURGE BINARY LOGS statement, 53, 105, 521
  - purge index file, 103, 105
  - purging binlog files, 103, 104
  - pvcreeate command, 566
  - pvsan command, 566
  - Python
    - adding relay servers, 172
    - additional information, 8
    - handling reporting, 46–48
    - handling switchovers, 134
    - PITR and, 575–579
  - PYTHONPATH environment variable, 619
- ## Q
- queries
    - analyzing, 160



- best practices, 327, 444, 526
  - cross-schema, 202
  - data sharding and, 193, 195, 202, 215, 218–220
  - data-mining, 43
  - distributing, 160
  - EXPLAIN statement and, 596
  - indexes and, 387
  - manually executing, 526
  - map-reduce execution and, 193
  - shard IDs and, 218
  - slave lag and, 498
  - troubleshooting, 505, 507, 511–512
  - Query Analyzer
    - described, 595, 605–608
    - enabling, 596
  - query cache
    - best practices, 441
    - described, 387
    - server variables, 388
    - status variables, 388
  - query events
    - binlog event structure, 55
    - context events and, 251
    - current database and, 116
    - described, 32
    - execution contexts, 59–64
    - interpreting, 114–115
    - logging, 59–65
    - mysqlbinlog support, 110
    - post header and body, 115
    - reading remote files, 111
    - row-based replication and, 280
    - statement-based replication and, 278
    - thread IDs and, 65, 254
  - Q\_AUTO\_INCREMENT status variable, 116
  - Q\_CHARSET status variable, 116
  - Q\_SQL\_MODE\_CODE status variable, 116
- ## R
- RAID (redundant array of inexpensive disks), 534
  - Rand event, 62, 252
  - RAND function
    - context events, 61
    - described, 60
    - Rand event and, 252
  - range mapping (data sharding)
    - adding new shards, 211
    - creating index tables, 211
    - described, 210
    - fetching shards, 212
    - reactive monitoring, 335
    - READ COMMITTED isolation level, 184
    - read-only option, 120
    - reading data
      - avoiding stale data, 183
      - load balancing and, 154
      - from raw binary log files, 112
      - on remote files, 111
      - scaling out and, 155
      - thread-local objects, 253
    - recovery images, 573, 577
    - recovery point objective (RPO), 542, 546
    - recovery time objective (RTO), 542, 547
    - redundancy (high availability)
      - MySQL Cluster and, 296–298, 324
      - principle overview, 6, 124–126
    - redundant array of inexpensive disks (RAID), 534
    - relay log
      - configuring slaves, 27
      - fsync calls and, 269
      - inspecting for slaves, 491
      - maintaining replication positions, 246–248
      - row event execution, 283
      - structure of, 229–232
      - troubleshooting, 515
    - relay log information file
      - described, 230, 232
      - fsync calls and, 269
      - manipulating slave threads, 234
      - replication status information, 246
      - storing replication information, 271
      - thread syncing and, 269
      - transactional replication and, 272
      - troubleshooting data loss, 514
    - relay servers
      - adding in Python, 172
      - handling failures, 127
      - hierarchal replication, 170
      - setting up, 171–172
      - synchronizing with, 181
    - relay-log option, 27
    - relay-log-index option, 27
    - relay-log-info-file option, 230
    - relay-log-space-limit option, 246
    - relay\_log\_info\_repository option, 271

- Reliability Monitor, 372–374
- remote files, reading, 111
- renice command, 343
- REORGANIZE PARTITION statement, 305
- REPAIR TABLE statement, 468
- REPEATABLE READ isolation level, 184
- Replicant Library (see MySQL Replicant Library)
- replicas, 297, 319
- replicate-do-db option
  - data sharding and, 204, 221, 225
  - replication monitoring, 479
  - slave filters and, 175
  - thread processing and, 250
- replicate-do-table option
  - replication monitoring, 480
  - slave filters and, 176
  - thread processing and, 250
- replicate-ignore-db option
  - replication monitoring, 479
  - slave filters and, 176
  - thread processing and, 250
- replicate-ignore-table option
  - replication monitoring, 480
  - slave filters and, 176
  - thread processing and, 250
- replicate-rewrite-db option, 480
- replicate-same-server-id option, 141, 250, 480
- replicate-wild-do-table option
  - replication monitoring, 480
  - slave filters and, 176
  - thread processing and, 250
- replicate-wild-ignore-table option
  - replication monitoring, 480
  - slave filters and, 176
  - thread processing and, 250
- replicate\_from function, 41
- replicate\_to\_position function, 134
- replication, 123
  - (see also high availability; row-based replication; scaling out; statement-based replication)
  - adding slaves, 35–42
  - architecture basics, 228–235
  - asynchronous, 6, 156–158
  - backup and recovery, 571
  - basic steps in, 24–29
  - best practices, 445
  - bidirectional, 140–144, 190
  - binary log example, 30–32
  - checking setup, 646–648
  - circular, 149–151, 158, 518, 684
  - common tasks, 612–618
  - common uses, 5–7, 154
  - configuration privileges, 26, 27
  - data sharding and, 193
  - described, 5
  - external, 319
  - filtering events, 174–176
  - GTIDs and, 266–267, 498–499, 519
  - handling broken connections, 248
  - hierarchical, 170–172
  - high availability and, 319–324
  - improving performance, 477
  - inclusive and exclusive, 478–480
  - internal, 319
  - managing topologies, 158–170
  - mixed-mode, 99
  - monitoring master servers, 483–486
  - monitoring slave servers, 487–492
  - multichannel, 322–324, 678
  - multisource, 275–278, 678, 687, 690
  - MyISAM storage engine and, 475
  - MySQL Cluster and, 320
  - MySQL Proxy and, 676
  - MySQL Workbench and, 493–494
  - pausing, 527
  - performing common tasks, 42–49
  - PITR and, 571
  - process overview, 29–35
  - repopulating tables, 676
  - reporting bugs, 528
  - running over Internet, 235–239
  - segmenting, 682
  - semisynchronous, 6, 136, 257–260
  - server setup and, 478
  - setting up, 644
  - setting up using GTIDs, 261–262
  - showing topologies, 648
  - slave safety and recovery, 268–275
  - slaves processing events, 249–256
  - status information, 239–248
  - stopping, 615–616
  - synchronous, 156–158
  - time-delayed, 684
  - tips and tricks, 675–692
  - transactional, 270
  - troubleshooting, 510, 517–521, 527

- REPLICATION CLIENT privilege, 28
- REPLICATION SLAVE privilege
  - reading remote files, 111
  - usage recommendations, 27–29, 73
- replication threads, 233, 481–482
- replication topology (see topologies)
- repopulating tables, 676
- report generation
  - cross-shard joins and, 193
  - performance considerations, 24, 155
  - process overview, 43–49
  - replication bugs, 528
  - scaling out and, 155
- report-host option
  - described, 240, 386
  - registering slaves on masters, 245
  - troubleshooting replication, 521
- report-password option, 240
- report-port option, 240
- REQUIRE SSL option, 512
- RESET MASTER statement
  - binlog file support, 53
  - clearing GTID variables, 499
  - described, 35, 616
  - GTID problems and, 519
  - slave promotion, 146
  - usage example, 35
- RESET SLAVE statement
  - described, 35, 616
  - master log information file, 231
  - usage example, 35, 221
- resource managers, 91
- restarts, best practices, 525
- restore process
  - after error replication, 572
  - commands supported, 550
  - expectations for, 545
  - forming archival plans, 546
  - LVM support, 567
  - MySQL Enterprise Backup, 557–559
- return values, stored functions and, 78
- ring topology, 523
- risk assessment, 539
- Role class
  - \_create\_repl\_user method, 20
  - described, 19
  - \_disable\_binlog method, 20
  - \_enable\_binlog method, 20
  - imbue method, 19
  - \_set\_server\_id method, 20
  - unimbue method, 19
- ROLLBACK statement, 86
- Romanenko, Igor, 560
- rotate events
  - binlog event structure, 53, 55
  - binlog-in-use flag and, 102
  - described, 32–33
  - header restrictions, 117
  - I/O threads and, 250
- round-robin DNS, 163
- round-robin multisource replication, 276
- Row class
  - \_\_getitem\_\_ method, 18
  - \_\_iter\_\_ method, 18
  - next method, 18
- row events
  - described, 278–281
  - execution of, 283–284
  - structure of, 282
- row-based replication
  - described, 30, 97–98, 278
  - enabling, 98
  - events and triggers, 284–286
  - filtering in, 286–287
  - logging statements, 58
  - mixed-mode replication and, 99
  - new events supported, 278–284
  - nontransactional changes and, 83
  - options for, 120
  - partial row replication, 288
  - query events and, 280
  - Table\_map event, 280
  - tips and tricks, 676
- rpl-semi-sync-master-enabled option, 258
- rpl-semi-sync-master-timeout option, 259
- rpl-semi-sync-master-wait-no-slave option, 259
- rpl-semi-sync-slave-enabled option, 258
- rpl\_semi\_sync\_master\_clients option, 259
- rpl\_semi\_sync\_master\_status option, 259
- rpl\_semi\_sync\_slave\_status option, 260
- RPO (recovery point objective), 542, 546
- RTO (recovery time objective), 542, 547

## S

- SAN (storage area network), 137
- sar command
  - described, 342, 350–352
  - installing, 344

- ksar tool and, 403
- Sarbanes-Oxley Act (SOX), 534
- savepoints, 438
- scaling out
  - asynchronous replication, 156–158
    - common uses, 154
    - data consistency and, 177
    - described, 6, 153
    - hierarchal replication, 170–172
    - managing replication topology, 158–170
    - reading data and, 155
    - specialized slaves, 173–177
    - writing data and, 155
- scaling up, 153
- scheduling tasks
  - on Unix, 48
  - on Windows, 48
- schemas
  - backing up, 225
  - best practices, 327
  - described, 190
  - Performance Schema feature, 409–421
  - shard IDs and, 204
  - sharded and global tables example, 198–202
- Schlossnagle, Theo, 8
- Schwartz, Baron, 8
- scripting clone operation, 41–42
- searches
  - Console application and, 360
  - metadata, 636
  - mysqldump utility and, 560
  - for processes, 637–639
  - row-based, 283
- secondary servers, 136
- Secure Sockets Layer (see SSL)
- security
  - binary log and, 73
  - information assurance and, 532
  - logfile messages, 370
  - monitoring considerations, 334
  - MySQL Workbench and, 398
  - password considerations, 72
  - replication threads and, 80
  - spawning processes and, 347
  - triggers and, 72
- SELECT MASTER\_POS\_WAIT function, 527
- SELECT statement
  - data consistency example, 184
  - data sharding example, 199, 215
  - EXPLAIN statement and, 423
  - LIKE clause, 384
  - LIMIT clause, 164
  - load balancing example, 164
  - logging considerations, 52
  - nontransactional changes and, 85
  - ORDER BY RAND() clause, 164
  - as read-only, 169
  - stored functions and, 79
  - troubleshooting memory tables, 513
  - troubleshooting queries, 511
  - WHERE clause, 441
- semisynchronous replication
  - configuring, 258
  - described, 6, 136, 257
  - monitoring, 259
- Server class
  - clone method, 41
  - connect method, 18
  - described, 17
  - disconnect method, 18
  - fetch\_config method, 18
  - replace\_config method, 18
  - sql method, 18
  - ssh method, 18
  - start method, 19
  - stop method, 19
- server clone utility, 639–641
- server IDs
  - configuring masters, 25
  - configuring slaves, 27
  - described, 25
  - dual-master setup and, 141
  - Role class and, 20
  - Server class and, 18
  - shared disks and, 137
  - troubleshooting, 510
- server roles
  - creating, 172
  - MySQL Replicant Library, 19–20
- server-id option
  - connection timeouts and, 510
  - described, 26
- server\_id option, 312
- SERVER\_STATUS\_AUTOCOMMIT flag, 218
- SERVER\_STATUS\_IN\_TRANS flag, 218
- SESSION keyword, 384
- SET statement
  - creating key caches, 473



- usage example, 73
- SET GLOBAL statement, 408
- SET TIMESTAMP statement, 108
- setup tables, 412
- SHA hash functions, 213
- shard IDs
  - in composite identifiers, 194
  - described, 206
  - queries and, 218
  - range mapping, 211
  - schemas and, 204
- shard management
  - described, 195, 220
  - moving shards to different nodes, 220–224
  - splitting shards, 225
- sharding (see data sharding)
- sharding indexes
  - automatically computing possible, 200
  - described, 197
  - hash mapping and, 214
  - multiple independent, 201
  - range mapping and, 211
  - schema example, 198–202
- sharding key
  - automatically computing possible sharding indexes, 200
  - dispatching queries, 218–220
  - mapping, 194, 206–215
  - schema example, 198–202
  - shard mapping functions and, 210–215
  - sharding schemes and, 206–209
- sharding technique (see data sharding)
- shell commands
  - common tasks, 685–688
  - managing replication, 13
  - Server class and, 18
- SHOW statement, 309, 312
- SHOW BINARY LOGS statement
  - described, 44, 386
  - monitoring master servers, 484
  - monitoring slaves, 490
- SHOW BINLOG EVENTS statement
  - context events and, 62
  - described, 385
  - error codes and, 86
  - hierarchal replication, 170
  - monitoring master servers, 484–486
  - monitoring slave servers, 491
  - troubleshooting replication, 521
- usage examples, 31, 34
- SHOW COLUMNS FROM statement, 424
- SHOW DATABASES statement, 410
- SHOW ENGINE INNODB MUTEX statement, 452
- SHOW ENGINE INNODB STATUS statement
  - described, 450–453
  - InnoDB monitors and, 454
  - monitoring buffer pools, 458
  - monitoring tablespaces, 461
  - troubleshooting with, 464
- SHOW ENGINE LOGS statement, 385
- SHOW ENGINE STATUS statement, 385
- SHOW ENGINES statement, 385, 437
- SHOW FULL PROCESSLIST statement, 403
- SHOW GRANTS FOR statement, 520
- SHOW INDEX statement, 384, 432
- SHOW INDEX FROM statement, 383
- SHOW MASTER LOGS statement
  - described, 386
  - monitoring master servers, 484
  - replication status information, 241
- SHOW MASTER STATUS statement
  - backup procedure, 575
  - best practices, 523
  - cloning the master, 37
  - data consistency example, 178–180
  - described, 386
  - master status variables and, 487
  - monitoring master servers, 483
  - pausing replication, 527
  - privilege considerations, 28
  - replication status information, 241
  - reporting bugs, 528
  - troubleshooting replication, 520
  - usage example, 133
  - usage examples, 34
- SHOW PLUGINS statement, 383
- SHOW PROCESSLIST statement
  - checking status, 614
  - described, 248, 383
  - monitoring slave lag, 496
  - monitoring threads, 481–483
  - mytop utility and, 403
  - processlist command and, 390
  - troubleshooting replication, 520
- SHOW RELAYLOG EVENTS statement, 386, 491

- SHOW SLAVE HOSTS statement
  - described, 386
  - replication status information, 240
  - slave status variables and, 492
  - troubleshooting replication, 521
- SHOW SLAVE STATUS statement
  - best practices, 523
  - cloning slaves, 39
  - data consistency example, 185
  - described, 386
  - monitoring lag, 496
  - monitoring slaves, 487–490
  - pausing replication, 527
  - privilege considerations, 28
  - replication status information, 242, 246
  - reporting bugs, 528
  - slave promotion, 148
  - slave provisioning, 618
  - slave status variables and, 492
  - status of replication in GTID positions, 262
  - transactional replication and, 272
  - troubleshooting replication, 520
  - troubleshooting slaves, 510, 513, 515
  - usage example, 612–615
- SHOW STATUS statement
  - described, 382, 384
  - extended-status command and, 390
  - limiting output, 384
  - monitoring key cache, 472
  - monitoring slave servers, 487
  - MySAR system activity report, 403
  - mytop utility and, 403
  - reading variable values, 260
- SHOW TABLE STATUS statement, 384
- SHOW TABLES statement, 410
- SHOW VARIABLES statement
  - described, 382, 384, 408
  - limiting output, 384
  - monitoring key cache, 472
  - MySAR system activity report, 403
  - usage example, 386
  - variables command and, 390
- SHOW WARNINGS statement, 427
- show-progress option, 554
- show-slave-auth-info option, 240
- SHUTDOWN statement, 314
- shutdowns, best practices, 525
- single point of failure, 295, 315
- slave election, 651
- slave filters, 174–176
- slave lag
  - causes and cures for, 497–498
  - monitoring, 496
- slave promotion
  - considerations, 130, 144
  - GTIDs and, 148, 264
  - revised method, 147–149
  - traditional method, 145
- slave provisioning, 618
- slave servers
  - adding, 35–42, 617–618
  - candidate slaves, 651
  - causes and cures for lag, 497–498
  - checking status, 523, 612–615
  - cloning, 39–40
  - configuring, 27, 38
  - connecting to masters, 26–29
  - creating, 7, 128
  - database crashes, 268–270
  - delayed slaves, 6
  - events and, 81
  - filtering replication events, 174–176
  - handling failures, 127, 130
  - hierarchal replication, 170
  - managing lag, 496
  - monitoring, 487–492
  - monitoring thread status, 482
  - partitioning events, 176
  - processing events, 249–256
  - removing from topologies, 130
  - replication overview, 5
  - safety and recovery, 268–275
  - scaling out and, 173–177
  - scripting the clone operation, 41
  - server roles, 19–20
  - status variables and, 492
  - synchronizing, 145, 156, 268–270
  - tips and tricks, 680–682
  - transactions and, 268–270
  - troubleshooting, 509–517
  - two-phase commit and, 156
  - upgrading, 130
- slave threads, 233
  - (see also I/O threads; SQL threads)
  - described, 228, 233
  - starting and stopping, 234
- slave-net-timeout option, 248
- slave-sql-verify-checksum option, 57

- SlaveNotRunningError exception, 16
- slave\_compressed\_protocol variable, 495
- Slave\_heartbeat\_period status variable, 492
- Slave\_last\_heartbeat status variable, 492
- Slave\_open\_temp\_tables status variable, 492
- Slave\_received\_heartbeats status variable, 492
- Slave\_retried\_transactions status variable, 492
- Slave\_running status variable, 492
- snapshots
  - backup comparisons, 569
  - backup operations, 304
  - described, 564
  - logical volumes and, 566
  - LVM support, 564
  - methods for taking, 128
  - System Profiler support, 357–359
- Solaris class, 17
- Solaris operating system
  - MySQL Replicant Library and, 12, 17
  - snapshot support, 129
  - ZFS support, 40, 568
- SOX (Sarbanes-Oxley Act), 534
- splintering (see data sharding)
- split-brain syndrome
  - described, 136
  - DRBD and, 140
  - MySQL Cluster and, 298
  - shared disk solution, 138
- SQL Editor, 400–402
- SQL nodes
  - external replication and, 321
  - starting, 311–313
- SQL threads
  - checking status, 248
  - context events, 251–253
  - described, 228, 233
  - filtering and skipping events, 255–256
  - processing overview, 249, 250–256
  - replication and, 481
  - starting and stopping, 234, 277, 615
  - state considerations, 243–246
  - synchronizing, 269
  - thread-specific events, 253
- SQL\_SLAVE\_SKIP\_COUNTER variable
  - described, 255, 271
  - troubleshooting tips, 505, 507
- ssh tunnel mode, 235
- SSL (Secure Sockets Layer)
  - configuring connection, 495
  - master log information file, 231
  - MySQL support, 73
  - replication over Internet, 235–237
  - troubleshooting slaves, 512
- ssl-capath option, 237, 512
- ssl-cert option, 237, 512
- ssl-key option, 237, 512
- star topology, 522
- START SLAVE statement
  - connecting master and slave, 28
  - described, 615
  - manipulating slave threads, 234
  - relay log information file, 232
  - restarting slave threads, 277
  - slave provisioning, 618
  - slave status variables and, 492
  - troubleshooting replication, 520
- START SLAVE IO\_THREAD statement, 234
- START SLAVE SQL\_THREAD statement, 235
- START SLAVE UNTIL statement
  - MASTER\_LOG\_FILE option, 133, 616
  - MASTER\_LOG\_POS option, 46
  - MASTER\_POS\_WAIT function and, 221
  - RELAY\_LOG\_FILE option, 616
  - SQL\_AFTER\_GTIDS option, 616
  - SQL\_BEFORE\_GTIDS option, 616
- START TRANSACTION statement, 86, 216–218
- start\_trans function, 179, 184
- statement-based replication
  - described, 30, 120
  - filtering and, 287
  - logging statements, 58–65
  - logging transactions, 88
  - partial execution of statements, 287
  - query events and, 278
  - special constructions, 82
  - tips and tricks, 676
- statement-end flag, 280
- statements (see logging statements)
- static sharding, 206–208
- statistics, monitoring, 355–356, 606
- status command, 390
- STATUS statement, 310
- status variables
  - described, 116, 381
  - InnoDB storage engine, 457, 459
  - monitoring master servers, 487

- monitoring semisynchronous replication, 259
  - monitoring slave servers, 492
  - MySQL Workbench example, 394
  - query cache, 388
  - query events and, 116
  - reading, 382
  - showing, 382
  - threads and, 386
  - Stop event, 104, 249
  - STOP SLAVE statement
    - described, 615
    - manipulating slave threads, 234
    - RESET SLAVE statement and, 35
    - slave promotion, 146
    - slave status variables and, 492
    - troubleshooting replication, 520
    - usage examples, 35, 44
  - STOP SLAVE IO\_THREAD statement, 234, 277, 615
  - STOP SLAVE SQL\_THREAD statement, 235, 277, 615
  - storage area network (SAN), 137
  - storage engines, 447
    - (see also specific storage engines)
    - best practices, 436–441
    - default, 677
    - monitoring, 447–475
    - troubleshooting, 511
  - stored functions
    - DEFINER clause, 78
    - described, 75, 78
    - INSERT statement and, 79
    - logging statements, 78–81
    - privileges and, 80–81
    - return values and, 78
    - SELECT statement and, 79
    - specifying characteristics, 78
    - SQL SECURITY INVOKER characteristic, 81
  - stored procedures
    - committing transactions and, 698
    - DEFINER clause, 77, 117
    - described, 75
    - logging statements, 75–78
    - replication tips and tricks, 688
  - stored programs
    - described, 70
    - handling events, 81
    - logging statements, 70–75
  - stored routines
    - DEFINER clause, 75
    - described, 75
    - logging statements, 70–75
  - string data, interpreting, 114
  - stunnel command
    - described, 236
    - replication support, 238–239
  - Sun Microsystems, 568
  - SUPER privilege
    - administrator account and, 394
    - configuring replication, 28
    - disabling, 119
    - logging statements and, 74
    - setting thread IDs, 65
    - SHOW PROCESSLIST statement and, 384
    - stored functions and, 81
  - svadm command, 17
  - swapping technique, 337
  - switchover, defined, 651
  - switch\_to\_master function, 134
  - Sync mutex, 95
  - sync-binlog option, 101, 119, 504
  - synchronizing
    - coordinating for filesystems, 101
    - I/O threads, 269
    - relay servers, 181
    - slave servers, 145, 156, 268–270
    - SQL threads, 269
    - troubleshooting, 513
  - synchronous replication
    - asynchronous replication and, 156
    - performance considerations, 157
  - sync\_master\_info option, 274
  - sync\_with\_master function, 179, 185
  - SYSDATE function, 61
  - sysstat package, 344
  - System Health Report, 366–369, 374
  - System Profiler, 357–359
  - system recovery (see disaster recovery)
- ## T
- table IDs, 282
  - tables, 62
    - (see also AUTO\_INCREMENT attribute; MyISAM storage engine; temporary tables)
    - assigning to different masters, 141–144



- autoincrement considerations, 97
- best practices, 526
- CACHE INDEX statement, 473
- checking indexes, 635
- compressing, 462, 468, 471
- data sharding and, 198, 204
- defragmenting, 471
- executing triggers against, 72
- global, 192
- InnoDB table monitor, 455, 462
- locking, 40
- nontransactional changes and, 83–86, 88
- repairing, 468
- repopulating, 676
- security considerations, 73
- setup, 412
- slave lag and lock contention, 498
- storing in index order, 470
- troubleshooting, 465, 505, 513–514
- tablespaces
  - described, 450
  - InnoDB tablespace monitor, 456, 460
  - logs as separate, 449
  - monitoring, 460
  - troubleshooting, 465
- Table\_map event, 278–281
- tar utility, 562
- Task Manager, 374
- Task Scheduler, 48
- temporary tables
  - process IDs and, 64
  - pseudothread IDs and, 254
  - thread IDs and, 64
  - troubleshooting, 513
- TEMPTABLE view, 142
- testing clusters, 313
- thrashing, 338
- thread IDs
  - described, 64
  - logging queries, 60
  - post headers, 115
  - TLS support, 254
- thread-local store (TLS), 254
- threads
  - binary log group commit process and, 94
  - dump, 228
  - replication, 233, 481–482
  - security considerations, 80
  - slave, 228, 233–235, 243–246
  - status variables and, 386
  - transaction cache and, 87
  - troubleshooting, 516
- timers, defined, 412
- timestamps
  - calculating slave lag, 496
  - detecting misdirected writes, 631
  - logging statements, 60–61, 70
  - mysqlbinlog support, 111
  - setting for events, 108
- Tkachenko, Vadim, 8
- TLS (thread-local store), 254
- top command, 341–344, 374
- topologies
  - best practices, 521–523
  - checking server status, 523
  - circular replication, 149–151, 158
  - defining, 13–15
  - dual-master setup, 6, 12, 135–144, 158
  - hot standby, 23, 130–134
  - managing, 158–170
  - multimaster issues, 518
  - removing slaves from, 130
  - showing, 648
  - tree, 12, 158
- tps (transactions per second), 352
- transaction cache, 87–91
- transaction coordinator, 304
- transaction identifiers, 260
- transaction managers, 91
- transactional replication
  - described, 270
  - details of, 272
  - setting up, 271
- transactions, 268
  - (see also nontransactional changes)
  - asynchronous replication and, 156
  - best practices, 526
  - data sharding and, 195, 215–218
  - epochs and, 297
  - GTIDs and, 697
  - logging, 86
  - MySQL Cluster and, 304
  - semisynchronous replication and, 257
  - slave servers and, 268–270
  - stored procedures and, 698
  - troubleshooting, 507–509, 515
  - two-phase commit and, 157
- transactions per second (tps), 352

transaction\_allow\_batching option, 327  
tree topology, 12, 158

## triggers

- creating, 72
- DEFINER clause, 117
- events and, 284–286
- invoking, 75
- logging statements, 70–75
- security considerations, 72

## troubleshooting

- best practices, 521–528
- binary log, 506
- binary log events, 503–505
- data loss, 514
- detecting misdirected writes, 630
- error handling, 517
- I/O threads, 515
- InnoDB storage engine, 464–467
- master servers, 503–509
- memory, 513
- memory tables, 503
- nontransactional changes, 507–509, 515
- queries, 505, 507, 511–512
- relay log, 515
- replication, 510, 517–521, 527
- server IDs, 510
- slave servers, 509–517
- storage engines, 511
- synchronization, 513
- tables, 465, 505, 513–514
- temporary tables, 513
- threads, 516
- transactions, 515
- unsafe statements, 507–509

two-phase commit, 156

## U

UAC (User Account Control), 48, 366  
UDFs (user-defined functions), 97, 252  
umount command, 566  
--uncompress option, 554  
underscore (`_`), 175  
UNIV\_DEBUG directive, 453  
Unix operating system

- automated monitoring, 356
- disk usage, 350–353
- general system statistics, 355–356
- memory usage, 347–349
- monitoring, 334, 341–356

MySQL Replicant Library and, 12  
network activity, 353–355  
process activity, 342–347  
scheduling tasks on, 48

UNIX\_TIMESTAMP function, 60–61

UNLOCK TABLES statement, 568

## UPDATE statement

- invoking triggers, 72
- LIMIT clause, 97, 287
- logging, 58
- nontransactional changes and, 275
- stored procedures and, 76
- troubleshooting memory tables, 513
- WHERE clause, 52, 59

Update\_rows event, 279, 283

uptime command, 340, 341, 355

## USE statement

- current database, 69
- usage example, 108

User Account Control (UAC), 48, 366

## user accounts

- cloning users, 642
- connecting master and slave, 28
- MEM installation and, 588
- MySQL Workbench and, 397

User class, 16

USER function, 97, 99

user-defined functions (UDFs), 97, 252

## User\_var event

- described, 62, 251
- mysqlbinlog support, 108–110

UUID, 194, 260

UUID function, 99, 507

## V

Vagabond role, 20

validate command, 550

## variables

- configuring servers, 381
- password considerations, 72
- query events and, 59
- thread-specific results, 254

variables command, 390

--verbose option, 618, 675

verification procedures, 539

vgcreate command, 566

vgscan command, 566

## views

- best practices, 441

- optimizing, 142
- virtual shards, 202–205
- vmstat command, 341, 353, 355
- volume groups, 565
- Volume Shadow Copy, 564

## W

- wait\_for\_pos function, 179, 184
- wait\_for\_trans\_id function, 184
- WAIT\_UNTIL\_SQL\_THREAD\_AF -  
TER\_GTIDS function, 187, 264
- WHERE clause
  - DELETE statement, 52, 275
  - EXPLAIN statement and, 426
  - SELECT statement, 441
  - UPDATE statement, 52, 59
- wildcards
  - mysqlbinlog support, 110
  - slave filters and, 175
- Windows Experience report, 366
- Windows operating system
  - Cygwin and, 563
  - Event Viewer, 369–372
  - monitoring, 334, 365–377
  - MySQL Replicant Library and, 12
  - Performance Monitor, 375–377
  - Reliability Monitor, 372–374
  - scheduling tasks on, 48
  - System Health Report, 366–369, 374
  - Task Manager, 374
  - Volume Shadow Copy, 564

- Windows Experience report, 366
- worst case scenario, 538
- Write\_rows event, 279, 283
- writing data
  - data sharding and, 156
  - detecting misdirected writes, 630
  - load balancing and, 154
  - scaling out and, 155
  - thread-local objects, 253

## X

- X/Open Distributed Transaction Processing  
model, 91–94
- XA protocol, 91–94, 101
- Xid event, 91, 110
- XtraBackup
  - backup comparisons, 569
  - cloning slaves, 41
  - described, 569
  - snapshot support, 129
- XtraDB storage engine, 569

## Z

- Zaitsev, Peter, 8
- Zawodny, Jeremy D., 403
- ZFS filesystem
  - backup comparisons, 569
  - performing backups, 568
  - snapshot support, 40, 129

## 关于作者

Charles A. Bell 博士是 Oracle 的高级软件工程师。目前是备份首席开发员，并且是 MySQL 备份和复制小组的成员。他同爱妻住在弗吉尼亚州的一个小镇。2005 年，他获得弗吉尼亚州立联邦大学的工程学博士学位。他的研究兴趣包括：数据库系统、版本控制、语义网络和敏捷软件开发。

Mats Kindahl 博士是 Oracle MySQL 小组的首席高级软件开发员。他是 MySQL 基于行的复制及其他几个复制功能的主要架构师和实现者，目前是 MySQL 高可用性小组的架构师和项目主管，正在开发 MySQL Fabric。在加入 MySQL 之前，他研究过形式化方法、程序分析、分布式系统，并且获得计算机科学的博士学位。他还做过几年的 C/C++ 编译器的开发。

Lars Thalmann 博士是 MySQL 复制和备份的开发经理，负责这些功能的策划和开发，并带领相应的团队。Thalmann 自 2001 年起就开始做 MySQL 开发，那时他是 MySQL 集群的软件开发员。最近，他创建并发展了 MySQL 的备份功能，从 2004 年起就引导了 MySQL 复制的变革，已经成为 MySQL 集群复制发展的重要角色。Thalmann 拥有瑞典乌普萨拉大学的计算机科学博士学位。

## 封面介绍

本书封面上的动物是美洲知更鸟（*Turdus migratorius*）。从它特有的黑色脑袋、泛红的橘色胸脯，以及棕色的背部，能很容易辨识出来。作为画眉家族的一员，它是最常见的美洲鸟之一。（虽然它同知更鸟的名字一样，知更鸟也有泛红的胸脯，但这两个品种并没有什么关系。）

美洲知更鸟分布在北美六百万平方米的区域范围，全年栖息在美国的大部分地区。通常认为它可以报春，清早和晚上都唱歌。它们吃无脊椎动物（通常是蚯蚓）、水果和浆果类。它们喜欢开阔、草较少的地方，所以它们常常出现在后院、公园、花园和草地等地方。

封面图片取自 *Johnson's Natural History, Volume II*。



# 高可用MySQL (第2版)

服务器瓶颈和故障是任何数据库部署中的常见问题，但并不一定会导致全面故障。这本讲实践的书解释了复制、集群和监控功能，无论MySQL系统运行在硬件、虚拟机还是云上，都能帮助你保护MySQL系统不会中断运行。

这本书由这些工具的设计者编写，揭示了关于MySQL可靠性和高可用性的一些不成文的或难以发现的问题，这些知识对于任何使用这个数据库系统的组织来说都非常重要。第2版描述了很多MySQL工具的变化。本书涵盖了全部5.5版本的知识，以及若干5.6版本的功能。

- 学习复制的基础知识，包括二进制日志和MySQL Replicant库的使用
- 通过冗余处理失效组件
- 使用横向扩展以管理读负载的增加，使用数据分片处理大型数据库和写负载的增加
- 在MySQL集群中的某个节点上存储并复制数据
- 监控数据库的行为、性能和关键操作系统参数
- 跟踪master和slave，处理它们的故障、重启、崩溃及其他事故
- 检查工具，包括MySQL企业监控器、MySQL实用工具、GTID等

**Charles A. Bell**博士是Oracle的高级软件工程师。目前是备份首席开发人员，并且是MySQL备份和复制小组成员。

**Mats Kindahl**博士是Oracle MySQL小组的首席高级软件开发员。他是MySQL基于行的复制及其他几个复制功能的主要架构师和实现者，目前是MySQL高可用性小组的架构师和项目主管，正在开发MySQL Fabric。

**Lars Thalmann**博士是MySQL复制和备份的开发经理。他创建并发展了MySQL的备份功能，引导了MySQL复制的变革，已经成为MySQL集群复制发展的重要角色。

**Strata**  
Making Data Work

Strata是一个由人、工具和能够将大数据转化为明智决策的技术构成的新兴生态系统。访问[oreilly.com/data](http://oreilly.com/data)可获得更多信息和资源。

DATABASES

图书分类：数据库

策划编辑：张春雨

责任编辑：刘舫



**Broadview®**  
WWW.BROADVIEW.COM.CN

O'Reilly Media, Inc. 授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

ISBN 978-7-121-26688-1



9 787121 266881 >

定价：128.00元